Quantised Distillation for Resource-Efficient Image Generation Hardware

Jake Davies

Bachelor of Science in Computer Science
The University of Bath
2024

Quantised Distillation for Resource-Efficient Image Generation Hardware

Submitted by: Jake Davies

Abstract

The visual quality of images generated by deep learning models is rapidly improving as neural network architectures are becoming more complex. This complexity comes at the cost of more compute at generation time, leading to these models being unsuitable for real-time video processing. In this paper, we investigate how quantisation and knowledge distillation can be combined to reduce the size of custom hardware for accelerating image generation. To do this, we train a series of quantised neural networks, and measure how distilling knowledge from a higher-precision teacher affects our ability to decrease the precision of our students. Additionally, we directly map our resulting models to custom hardware descriptions, and evaluate the economics of creating such hardware. We find that when combining these techniques, students with smaller architectures than their teachers experience far greater benefit from knowledge distillation, even when the smaller architecture initially outperforms the larger architecture. Furthermore, we demonstrate that for even small models, direct mapping to hardware does not produce economically viable hardware. In addition to informing future work combining these techniques, our findings provide valuable insight for those looking to commercialise the mapping of image-generating networks to hardware.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost Ken Cameron, my supervisor, for being the only reason this project has surmounted to any value. His expertise in computer hardware, and the guidance he has provided throughout this project has been invaluable.

I would like to also thank my parents for their continued support into my education, and my friends for the motivation and competition they provide for me.

Chapter 1

Introduction

Modern computer architectures feature a variety of hardware components which are specialised for performing specific tasks, known as accelerators. It is common for even personal computers to have dedicated hardware for rendering, deep learning, video decoding, and more (Clover, 2023). This design allows the Central Processing Unit (CPU) to offload work directly to these accelerators and run in parallel with them to increase computer performance.

Accelerators are different to the CPU or Graphics Processing Unit (GPU) as they are designed to run one particular algorithm. Whilst they may be programmable, they are often not Turing Complete; they are designed for a specific problem domain. Because these domain-specific architectures are so specialised, they can offer a much lower latency, power consumption, and hardware size than running code on the CPU or GPU (Dally, Turakhia and Han, 2020).

There are two primary ways to deploy hardware accelerators, Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). Whilst ASICs are more efficient, they are fixed in functionality, and are designed for production at massive scale. They are created by fabricating the circuit design onto a silicon wafer, possibly with Peripheral Component Interconnect Express (PCIe) interfaces if designed to be used with motherboards.

On the other hand, FPGAs are reprogrammable, meaning the hardware design can be loaded onto them and adjusted dynamically, making them far cheaper and practical for prototyping at the cost of some efficiency. An overview of the hardware space is shown in Figure 1.1.

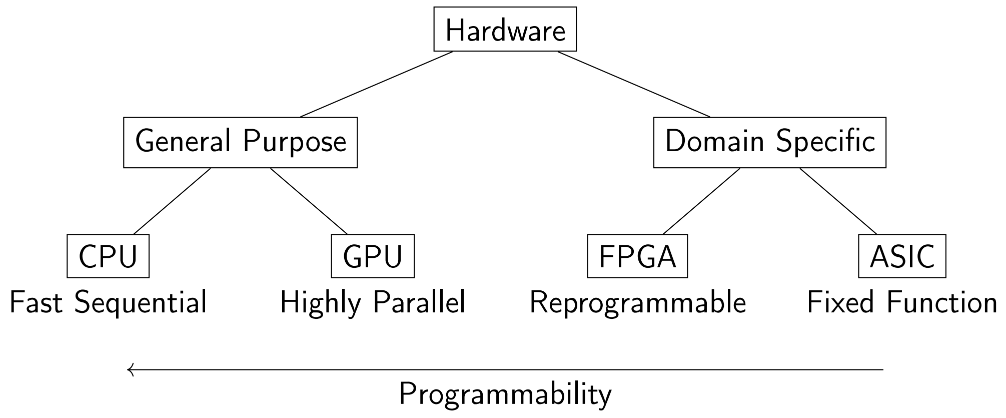


Figure 1.1: Hierarchy of hardware (Han, 2017)

Alongside this recent change in design philosophy, there have been significant advancements in deep learning which have produced extremely impressive results. Computer vision is one area that has seen significant benefit from data-driven approaches in both the breadth of problems that can be solved, and the visual quality of the solutions (Chai et al., 2021).

Whilst the results of these solutions can be vastly superior, they are often very compute intensive and slow to run. Image-generating models in particular suffer from this, because the final result must be a tensor (multi-dimensional matrix) with, for example, 6 million floating point numbers for a 1080p image. Other deep learning models can often reduce the representation through each layer, ending up at a far smaller number of possible classifications.

Custom hardware for accelerating model inference (image generation) exists, but it is usually designed for massive-scale processing in the cloud, as opposed to on-device inference (Wang, Wei and Brooks, 2019). This limits the suitability for this hardware for real-time, low-cost use. This is a problem as image-generation and image-to-image translation problems have many practical use cases. For example, image super-resolution could be used to upscale content to arbitrarily high resolutions. To be practical for consumers, it would require real-time inference and high-throughput processing on-device, which is challenging for deep learning models.

Even models designed for fast inference on dedicated deep learning hardware struggle to perform super-resolution above 25 frames per second (fps) (Ignatov et al., 2021). Clearly, general deep learning accelerators are not powerful enough to keep up with complex model architectures. To investigate this, we began by asking the following research question:

Does image generation benefit from hardware acceleration?

1.1 Contributions

The aim of this research is to explore ways of accelerating the inference step with hardware, to enable the development of low-cost, image-generating hardware for consumers. We claim that this is currently impossible through the direct mapping of deep learning models to hardware. We have found that whilst mapping is possible, it results in huge hardware sizes for even relatively small models, leading to very expensive hardware at GPU performance.

Our primary contribution comes from the combination of two compression techniques for reducing the size of deep learning models. Whilst the combination of these techniques has been explored before, our research reveals that they must be combined in a specific way in order to experience positive results. In addition, we believe we are the first to provide an economic perspective of this problem. This work allows researchers to make more informed decisions when combining these compression techniques for image-generation, and allows companies to more intelligently explore the production of dedicated hardware for image-generation.

Having introduced the problem area, we will now discuss the background of this problem space, including convolutional neural networks and hardware development. We then provide a review of the literature in this space, exploring image-generation models, image-similarity metrics, hardware support for deep learning, and case studies of other deep learning hardware. Having reviewed the literature, we move on to describe the methodology we undertook, including our hypothesis, experiments, and implementation details for this project. We finally evaluate the results of our experiments, discuss some of the problems with our approach, and conclude describing what we have learnt, and potential areas of future work.

Chapter 2

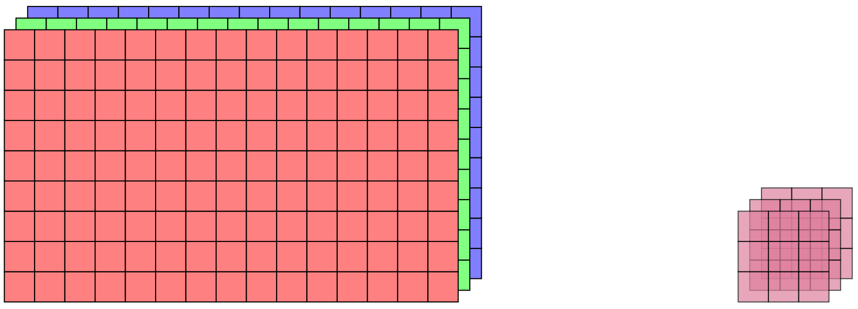
Background

In the following pages, we will introduce the key building blocks of image-generating models, highlighting some of the terminology used in this paper. We will also provide a brief introduction to hardware development, providing the rationale for designing custom hardware as method for accelerating deep learning.

2.1 Convolutional Neural Networks

The Deep Convolutional Neural Network (CNN) is a computational model which has allowed for significant advances to be made in computer vision and image processing. These models consist of convolutional layers that, when chained together, are able to represent any continuous mathematical function (Cybenko, 1989; Hornik, 1991; Zhou, 2020). These models are able to approximate functions through training on examples, requiring little human intervention.

These models are a variation of the Multi-Layer Perceptron (MLP) model, but they learn convolution kernels (filters) which operate on images to incorporate spatial information (LeCun et al., 1989; Lecun et al., 1998). Unlike in image processing, a 3×3 convolution in deep learning is actually a $3 \times 3 \times c$ convolution with $9c$ parameters, shown in Figure 2.1.



(a) RGB image

(b) $3 \times 3 \times 3$ filter

Figure 2.1: Convolutional filters in deep learning

Deep convolutional networks feature a series of layers, where each layer learns a set of filters that operate on the output of the previous layer. After an image has been fed into the network, we call the output of each layer a feature map, with multiple features (channels). Each feature is produced by convolving a filter from one layer across the whole feature map from the previous layer, and then adding a bias term which shifts the final value. The network can also feature layers to perform downsampling, upsampling, or reshaping (Shi et al., 2016). The parameters that make up each filter are learnt by the model during training.

Training these models involves a few steps. After initialisation, the first step is to feed the model an image and observe its output, which is known as performing a forward pass, or inference. We then compare that output with the output we expected using a loss function. Loss functions quantify the difference between the tensor the model produced, and the tensor we wanted the model to produce, where this difference is known as the loss (denoted $\mathcal{L}$).

Once we have the loss tensor, we can determine how much each filter and feature in the last layer should be adjusted to get the output we want. We then pass those changes to the previous layer and so on until the start of our network. This algorithm is called backpropagation (Rumelhart, Hinton and Williams, 1986). We can visualise the loss as a function of our network's parameters, where these updates represent a step down the steepest gradient.

In theory, we should do this for each training example in our dataset, combine the adjustments, and then adjust all the trainable parameters in that way. We call each iteration over the full dataset a single epoch. In practise this is done for smaller mini-batches, to get more frequent updates, known as stochastic gradient descent. Parameters that we set before training, such as the mini-batch and initial step sizes, are referred to as hyperparameters.

Stochastic gradient descent is slow; other methods have been proposed as alternatives, called optimisers. The Adam optimiser is a particularly successful optimiser used in practise (Kingma and Ba, 2017). It combines the ideas of momentum and adaptive gradients. Momentum dynamically increases the step size when previous steps have been in similar directions. This speeds up gradient descent but introduces the possibility of huge steps which skip past the minima of our loss function (Polyak, 1964).

Adaptive gradients decays the step size gradually, decaying steeper directions first, by normalising with recent previous gradients (Duchi, Hazan and Singer, 2011; Hinton and Tieleman, 2012). This stops gradient descent overshooting too far in the wrong direction, having to make up for it with many smaller steps later. Adam combines these techniques and has been shown to make training converge much faster for standard networks (Kingma and Ba, 2017).

In software frameworks, neural networks are implemented as computational graphs (compute graphs) (Minhas, 2021). These compute graphs can be built dynamically during the forward pass, statically before training, or not built at all (Minhas, 2021). PyTorch is one such framework that builds the compute graph dynamically during each forward pass. Shown in Figure 2.2, these graphs are abstract syntax trees representing computations on tensors.

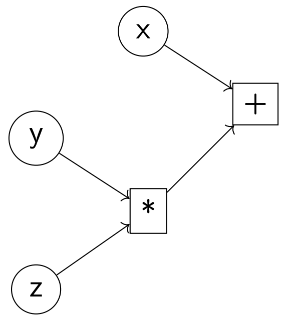


Figure 2.2: Computation graph for $x + y \times z$

2.2 Domain Specific Architectures

The CPU is the general-purpose chip that does the majority of the processing on a computer. Whilst the CPU is fast, being general purpose means it cannot be utilised completely for deep learning inference. Figure 2.3 shows us the breakdown of computations in a convolutional neural network for both CPUs and GPUs.

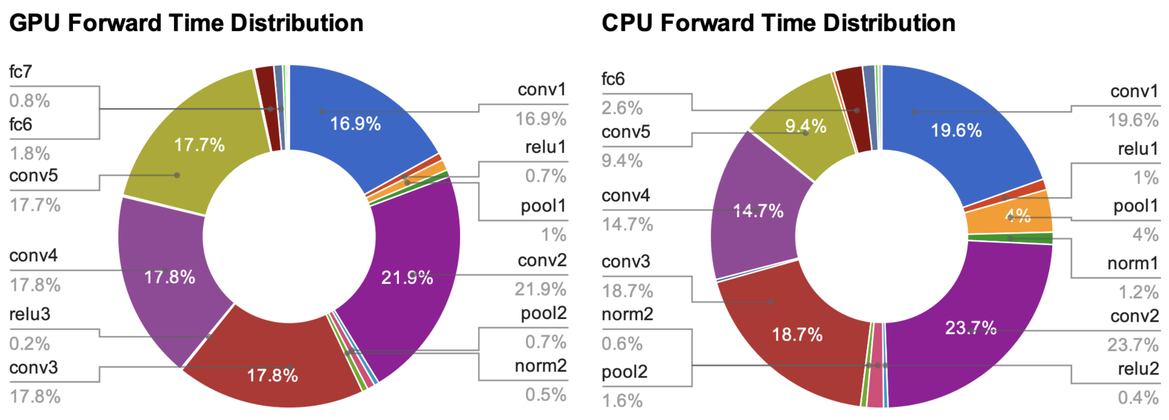


Figure 2.3: Convolutional neural network forward pass (Jia, 2014)

Figure 2.3 shows us that in a convolutional neural network, the computation time of the convolutions dominate across platforms. Unfortunately, for neural network computations, the Floating Point Unit and SIMD execution cores are utilised, which are highlighted in Figure 2.4. The remaining space on the chip is not very useful for deep learning.

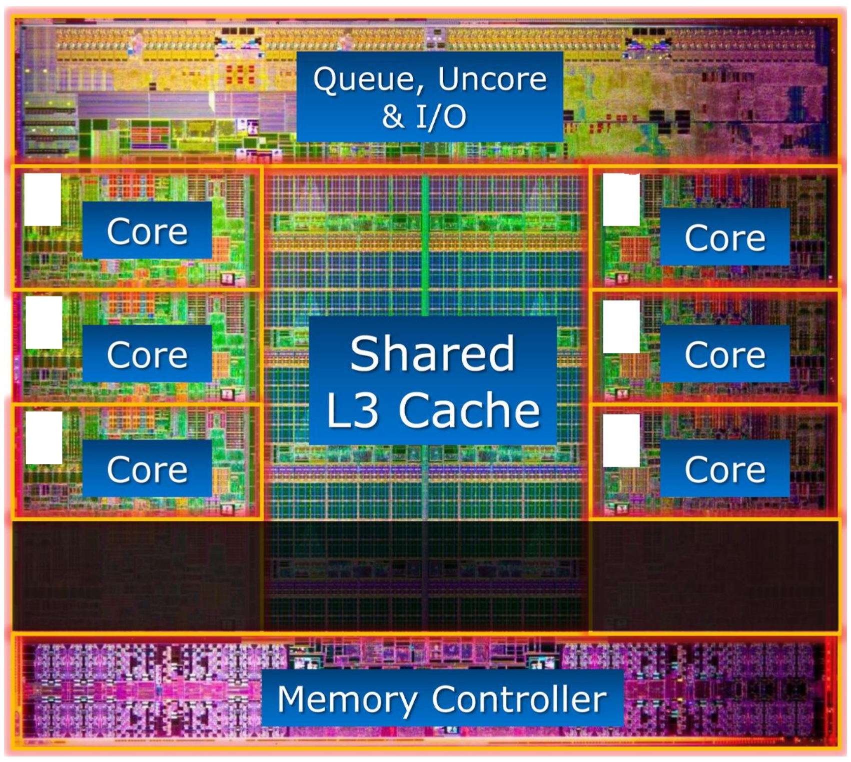
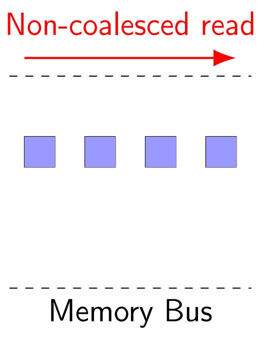
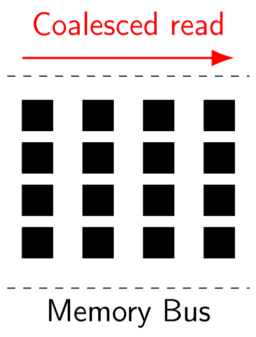


Figure 2.4: Intel Core i7-3960X, with execution units highlighted (Angelini, 2011)

Consumer GPUs typically have thousands of slower cores, compared to a standard 4-8 core consumer CPU (Ghorpade, 2012). Whilst they are not suitable for all problems, the throughput of these architectures can be so much greater for problems which are highly parallelisable. One feature that GPUs often implement is coalesced memory transactions. This is where memory that is read in a simple pattern can be combined into a single read, shown in Figure 2.5 (Davidson and Jinturkar, 1994). For example, if threads 1, 2, 3, 4 wanted to access the data in addresses 0x0, 0x4, 0x8, 0xc, the read could be done in a single transaction, given a wide enough memory bus.



(a) Non-coalesced read



(b) Coalesced read

Figure 2.5: Impact of memory coalescence

Coalesced transactions, along with many other features, have led to very efficient matrix multiplication operations on GPUs (Ma et al., 2019). In CNN implementations, the convolution operation is transformed into a matrix multiplication through the Im2Col transformation, shown in Figure 2.6. This is because convolutions have irregular memory accesses. Im2Col arranges memory such that the resulting matrix multiplication has regular memory reads (Nikam, 2017).

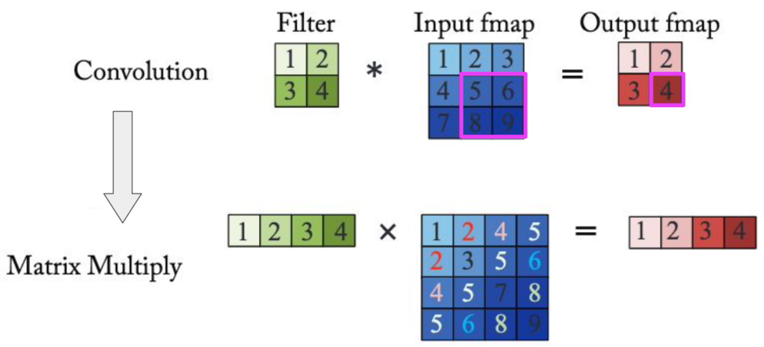


Figure 2.6: Im2Col transformation (Abdelfattah, 2022a)

Memory coalescing is a useful hardware feature, but we can do better. The Tensor Processing Unit (TPU) architecture supports convolutions directly in its instruction set (Kaz Sato, 2017). If we want to directly support an operation in hardware, we have to come up with a circuit design that performs it. This design can then be fabricated onto silicon, which creates a chip known as an Application-Specific Integrated Circuit (ASIC).

Traditionally, creating custom hardware alike the TPU has been an expensive process, requiring extensive testing and validation of functionality before fabricating the design on silicon. These chips can perform the task they are designed for in a fast and power-efficient manner, however they cannot perform any other operations, unlike general-purpose CPUs and GPUs.

Field-Programmable Gate Arrays (FPGAs) are a less efficient but far more flexible approach to hardware development. The high-level architecture of an FPGA is a combination of Block Random Access Memory (BRAM), along with an array of interconnected lookup tables (LUTs). BRAM is a larger store of memory, useful for communicating with other devices if the FPGA is part of a larger system-on-a-chip. These chips can include various input and output interfaces, as well as a microprocessor which controls the whole system (Figure 2.7).

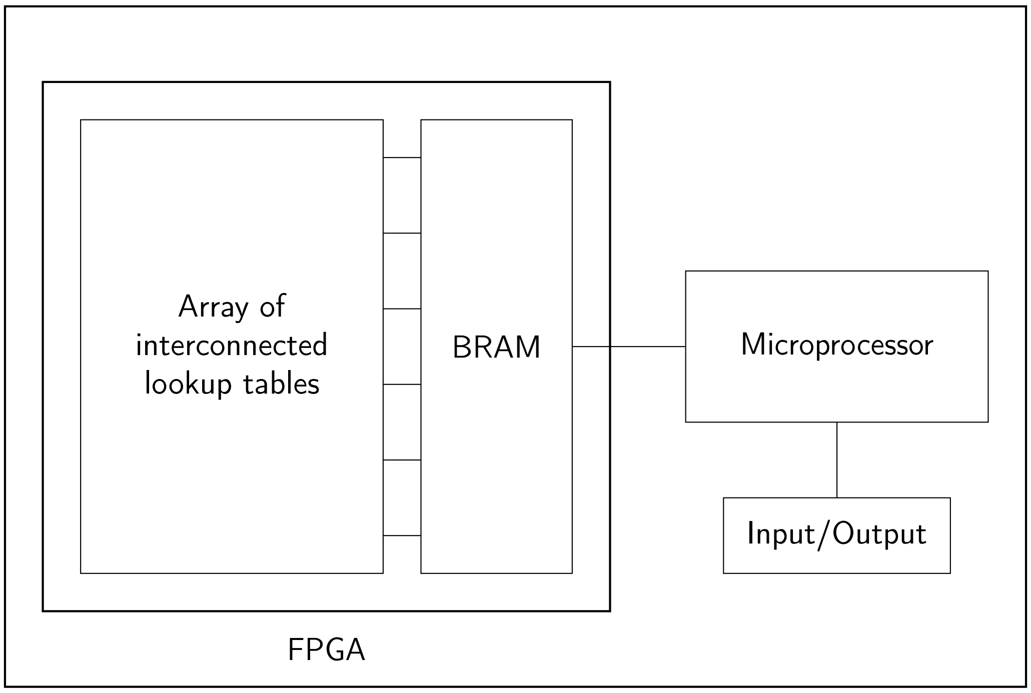


Figure 2.7: System-on-chip with integrated FPGA

Lookup tables are programmable components which implement *n*-input boolean functions. For example, Figure 2.8 shows a two-bit lookup table, where each of the configuration bits represents the single-bit output of the corresponding two-bit input. Configuration bit 0 is set to the output of "00", configuration bit 1 represents the output of "01", and so on.

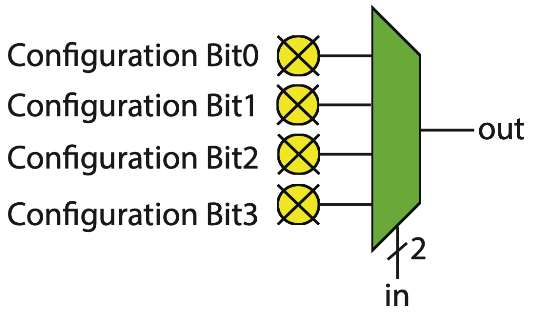


Figure 2.8: Two-bit lookup table (Kastner, Matai and Neuendorffer, 2018)

The number of lookup tables used in a circuit can be a measurement of how complex the design is (Kastner, Matai and Neuendorffer, 2018). What makes FPGAs such a flexible architecture is that these lookup tables, and the way in which they are interconnected, can be reprogrammed at any time after their manufacture, making them "field programmable".

To create hardware designs without working at the level of logic gates, we can describe our desired circuit using hardware description languages (Mead and Conway, 1980). Languages such as Verilog and VHDL allow the programmer to write higher-level code that will later be synthesised into hardware for a particular target architecture.

High-Level Synthesis (HLS) is a newer alternative to HDL programming which allows a developer to write in a higher-level language, such as C, C++, or SystemC (each with various restrictions). These programs can then be compiled into code written in a hardware description language, which can further be synthesised for a target.

When developing high-performance designs, there is a trade-off that has to be made between throughput and hardware area. For example, we could have 100 blocks that each can add two numbers together, allowing for 200 numbers to be added at the same time. This increases the overall throughput at the cost of having 100 blocks, which would require 100$\times$ the number of lookup tables. To save resources we could have one shared adder, but now the system can add only two elements at one time. This problem is known as the throughput-area trade off.

The area taken up by our design is important. This is because the cost of a silicon wafer is fixed, so if there are more usable chips produced from a single wafer, the design is cheaper to fabricate as an ASIC (Abdelfattah, 2022b). We call the number of usable chips produced from a single wafer, the yield. For FPGAs, we need to be able to fit our design into the number of lookup tables our FPGA can hold. This is an important consideration as whilst research may allow for arbitrarily large hardware, consumers want cheaper hardware.

Chapter 3

Literature, Technology, and Data Survey

The previous chapter introduced convolutional neural networks, the rationale for domain-specific architectures, and an introduction to custom hardware. This chapter introduces the research that has come about from the application and combination of these concepts, identifying trends and developments in deep learning hardware. We will cover image-to-image network architectures, the compression of deep neural networks, image similarity metrics, and case studies of successful hardware. Specifically, we will focus on the problem of super resolution.

3.1 Image Generation

Fully convolutional networks can be trained for image super-resolution without much adjustment (Dong et al., 2015). The quality of these networks was then improved by introducing residual connections, which allow earlier data representations to propagate deeper on in the network, shown in Figure 3.1 (He et al., 2016; Mao, Shen and Yang, 2016). Residual connections can add tensors element-wise, or concatenate them, depending on the application (Ras, 2018). Concatenation leads to larger networks due to the later filters having to become deeper.

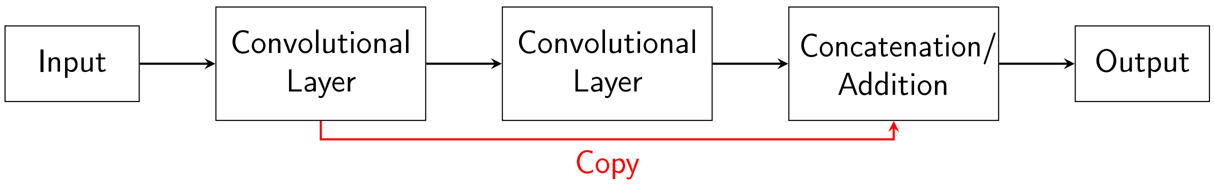


Figure 3.1: Residual connection (red) in a neural network

Previous attempts for super-resolution networks had to perform naive upsampling at the start of the network. This means for a 2x resolution, the network instantly becomes 4$\times$ more compute intense throughout. Instead of using a naive technique (such as bilinear or bicubic interpolation) at the start of the network, an efficient reshaping operation known as Pixel Shuffle was proposed. This operation is demonstrated in Figure 3.2. This reshaping layer allowed architectures to postpone upsampling until the end of the network (Shi et al., 2016). This architecture was called the Efficient Sub-Pixel Convolutional Network (ESPCN). Before Pixel Shuffle, the transposed convolution was used, however it was known to cause checkerboard artefacts, shown in Figure 3.3 (Odena, Dumoulin and Olah, 2016).

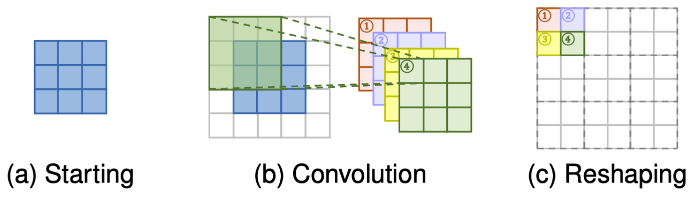


Figure 3.2: Pixel Shuffle operation (Wang, Chen and Hoi, 2020)

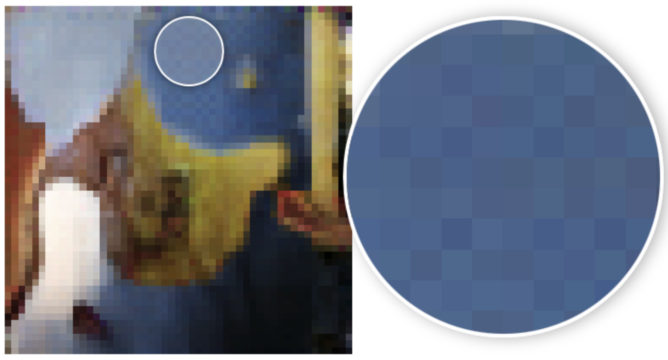


Figure 3.3: Checkerboard artefacts (Odena, Dumoulin and Olah, 2016; Dumoulin et al., 2017)

To improve upon these architectures, super-resolution networks were then trained using the Generative Adversarial Network (GAN) training methodology (Goodfellow et al., 2014). This is where two networks are trained, one that generates images, and one that classifies images as real or generated. In this architecture, we adjust only the loss function; the generator network is still just a convolutional network for inference. SRGAN is seen to be the first network that is able to upscale images 4$\times$ whilst preserving details (Ledig et al., 2017). GANs are notoriously difficult to train in a stable way, and can suffer from problems such as "mode collapse", where the generator finds a set of examples that fool the discriminator and just uses them until the discriminator is updated again (Goodfellow et al., 2014).

Diffusion models were later used to further improve the quality of image generated (Saharia et al., 2021). Diffusion models are a more stable method for training networks to produce images. They were first proposed as a probabilistic model, and later used to generate higher-quality images than the state-of-the-art GANs (Sohl-Dickstein et al., 2015; Ho, Jain and Abbeel, 2020). The idea of diffusion models is to progressively add noise to images from the training set, and try to teach the model to remove different levels of noise.

Diffusion models often use a U-Net architecture (Figure 3.4) which take in a noisy image and aim to output the noise that was added (Ronneberger, Fischer and Brox, 2015). That noise can then be subtracted from the original image to check. In reality the initial guess will be poor, so the network will add a majority of the noise back, and repeat this process iteratively.

Finally, transformer models have also shown great success for super-resolution, at the cost of huge parameter counts (Lu et al., 2022). Transformer models use a mechanism known as self-attention, which allows distinct image patches to communicate and provide each other with further semantic meaning (Dosovitskiy et al., 2021; Vaswani et al., 2023).

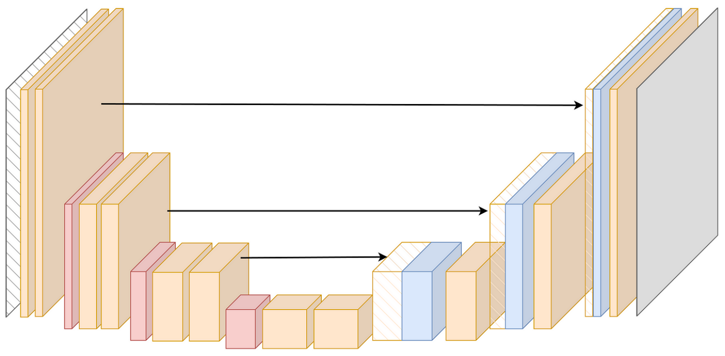


Figure 3.4: U-Net architecture (O'Sullivan, 2023)

When evaluating super-resolution performance, we want to compare the similarity of our generated image, with that of the desired output. For measuring the similarity between two images, the Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index Measure (SSIM) are two of the most widely used metrics (Pang et al., 2021). It has been argued that both of these metrics are quite poor at measuring the similarity of images, and can be tricked by distorted images (Zhang et al., 2018). Particularly, neither metric aligns with how humans perceive image similarity. The equations and their explanations for PSNR and SSIM can be found in Appendix A.1.

Learned Perceptual Image Patch Similarity (LPIPS) has been proposed as a more human-aligned and modern metric that does not suffer from the same flaws. The metric is computed by comparing the output vector of a trained neural network for both of the images. The neural network can be a simple neural network such as AlexNet (Krizhevsky, Sutskever and Hinton, 2017). Some qualitative comparisons to PSNR and SSIM are shown in Figure 3.5.

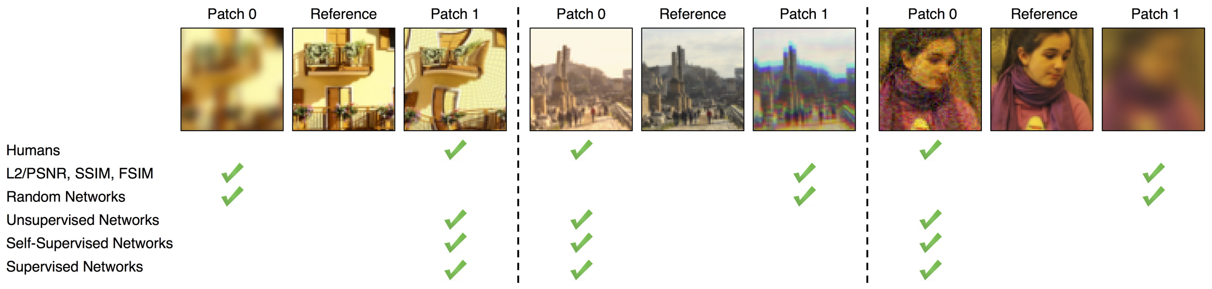


Figure 3.5: Comparison of image similarity metrics (Zhang et al., 2018)

From Figure 3.5 it is clear that both the PSNR and SSIM metrics are not accurate for images with heavy Gaussian blurring. Their study has shown that human annotators agree far more with the neural network-based metric (Zhang et al., 2018). The LPIPS score can also be used as a loss function during training (Zhang et al., 2018). Being the forward pass of another network, it is an expensive loss function to run.

Alternatives loss functions include the Mean Average Error (MAE, denoted $\mathcal{L}_1$), the Mean Square Error (MSE, $\mathcal{L}_2$), and variations of the SSIM metric. The $\mathcal{L}_1$ loss has been shown to converge the fastest for simple super-resolution networks (Zhao et al., 2018). The MSE and MAE loss are shown in Equation 3.1 and Equation 3.2, where θ is our model's current parameters, y_i is the ith target image, and $\hat{y}_i(\theta)$ is the output of the inference step.

$$\mathcal{L}_1(\theta) = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i(\theta)| \tag{3.1}$$

$$\mathcal{L}_2(\theta) = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}_i(\theta))^2 \tag{3.2}$$

There exist common datasets used to train and evaluate super-resolution networks. For training the DIV2K dataset of 1000 high-quality images is often used (Agustsson and Timofte, 2017). The Set5, Set14, BSD100, and Urban100 datasets are used as the test sets for evaluating the performance of trained networks (Bevilacqua et al., 2012; Zeyde, Elad and Protter, 2012; Martin et al., 2001; Huang, Singh and Ahuja, 2015). These datasets contain lower quality images of scenery, artwork, and buildings. There are a variety of other datasets with different focuses, however these are the most common and we will use them for comparability.

The speed of the inference step is another metric of interest. The two primary metrics for this are latency and throughput. The latency measures how long a single inference takes for our network, and the throughput measures the maximum inferences per second we can get out of our network, quantified in frames per second (fps). It is worth noting that 20 – 60 fps is considered real-time, which matches video playback (Gribbon, Johnston and Bailey, 2003; Schmittler et al., 2004; Shi et al., 2016; Kim, Choi and Kim, 2019; Sun et al., 2022).

GigaGAN is a network designed for text-to-image synthesis, but can also perform image super-resolution (Kang et al., 2023). The results are visually stunning, and the authors claim to be over 20$\times$ faster than the next best diffusion model. However, they state their inference time to still be 0.13 seconds for a small 512 $\times$ 512px image, and the paper makes no attempt to quantify the possible throughput.

Their measurements were also made on an NVIDIA A100 GPU, which is specialised for deep learning and costs over £10,000 (Dihuni, 2024). For an implementation without pipelining or parallelisation (overlapping computations), this leaves us with a poor 7.7 fps at images sizes of 512 $\times$ 512px on a £10,000 GPU. This clearly demonstrates the increasing importance of accelerating these processes for practical use.

3.2 Image Generating Hardware

Deep learning hardware often relies on software techniques being performed first to minimise computation time. Examples of this can be seen in Strassen's Matrix-Multiply Transform (Strassen, 1969), and the Winograd Transform (Winograd, 1978; Lavin and Gray, 2015). Strassen's transform replaces some of the multiply operations in a matrix-multiply into addition operations. The Winograd Transform is a way to produce the minimal possible number of multiplications for a convolution window, and can reorganise some into additions. These are important transforms because the energy cost of an addition is far cheaper than the cost of a multiplication in hardware (Horowitz, 2014).

For deep learning inference, a lot of the parameters are known in advance from training, and intermediate results can be pre-computed before deployment for both techniques. Neither technique is perfect; Strassen's transform performs better for only large matrix multiplications, and the Winograd Transform is specific to the filter and image sizes, and requires high-precision.

These are problems as convolutional networks use small filters. It has been shown that you can preserve the same receptive field by using smaller filters on more layers, as in the VGGNet architecture (Simonyan and Zisserman, 2015). Modern deep learning also uses low-precision data representations to save on compute, such as the TensorFloat16, BFloat16, FP16 (IEEE 16-bit Floating Point), and INT8 precisions (Gupta et al., 2015).

Custom instructions are also a way to perform deep learning computations more efficiently. In traditional instructions, there are overheads of memory accesses to L1 cache and the register file, along with the control of the instruction. These overheads take drastically longer than performing the operation (Abdelfattah, 2022b). Some of the instructions NVIDIA implement in their hardware are shown in Table 3.1.

Operation	**Energy (pJ)**	**Overhead**
Half-precision fused multiply-add	1.5pJ	2000%
Four-way dot product	6.0pJ	500%
16×16 matrix multiply	110pJ	27%

Table 3.1: Deep learning instructions (Dally, 2018)

Floating Point Operations Per Second (FLOPS) is often a metric used to compare the raw potential performance of deep learning hardware, calculated by Equation 3.3. Whilst an interesting statistic for comparison, in reality this metric is less useful as it requires full utilisation of the hardware the whole time. The NVIDIA A100 GPU specification lists multiple metrics, as the GPU supports a variety of number representations, shown in Table 3.2.

$$\text{FLOPS} = \frac{\text{floating point operations}}{\text{cycle}} \times \frac{\text{cycles}}{\text{second}} \times \begin{matrix}\text{number of processing}\\ \text{elements (multipliers)}\end{matrix} \tag{3.3}$$

Datatype	**Trillion Operations Per Second**
FP64	9.7
FP32	19.5
Tensor Float 32	156
BFLOAT16	312
INT8	624

Table 3.2: NVIDIA A100 metrics (NVIDIA, 2024)

Table 3.2 also shows us that NVIDIA support integer operations. Compared to floating points, integer operations are far more efficient to perform. Figure 3.6 shows the circuit design for a floating-point addition in comparison to the integer addition. This circuit is far more complex due to floating-points having to be mantissa-aligned before integer addition. Another benefit of using smaller precisions is that data movement becomes cheaper, as more data can fit into a single memory bus, such as in a memory coalescing read.

Integer operations are so useful, that there have been significant efforts to represent neural networks using only integer weights and biases. Whilst some parts of a neural network need to be in higher-precision (such as accumulators that store the results of dot products), the reduction of precision has been shown to be very successful (Han, Mao and Dally, 2016).

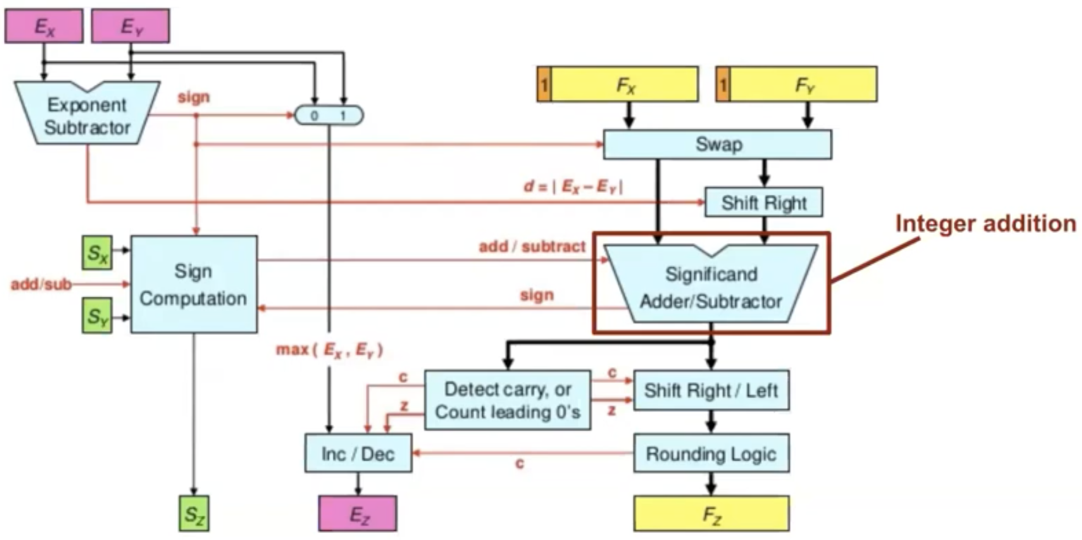


Figure 3.6: Floating-point adder (Abdelfattah, 2022b)

This technique is called network quantisation. Recently, quantisation been pushed to the extreme, with architectures such as single-bit super-resolution networks (Kim and Smaragdis, 2016; Ma et al., 2018). Quantisation is just a way of mapping from real numbers to integers so that we can represent our floating point networks with integer precision. The quantisation formula is described by Equation 3.4

$$Q_b(x) = clip(round(\frac{x}{s}) + z, 0, 2^b - 1) \tag{3.4}$$

$$where\ s = \frac{r_{max} - r_{min}}{2^b - 1}$$

where b is target number of bits, r represents the range of all inputs, and z is the number that zero would be mapped to (Abdelfattah, 2022c). We call s the scaling factor, and if $z = 0$ we say that the quantisation is symmetric (Abdelfattah, 2022c). The clipping and rounding functions each cause a source of error, and there is a trade-off between them. The method of rounding can also impact the final model accuracy and it has been suggested that the rounding technique should be a learnt parameter during training (Nagel et al., 2020).

Quantisation can reduce the overall computation during inference, as applying the scaling factor and shifting by the zero point can be folded into smaller computations once they have been set. Quantisation can be done during training, which is known as Quantisation-Aware Training (QAT), or post-training. QAT allows for improved accuracy, but as $Q_b(x)$ is non-differentiable, a fake quantised forward-pass through the network is used during training, whilst the FP32 representation is kept for backpropagation, increasing training times. Whilst previous work has encouraged the use of the $\mathcal{L}_1$ loss for training super-resolution networks, it is unclear whether this finding extends to quantised networks (Zhao et al., 2018).

Pruning is another deep compression technique that zeroes out some of the parameters in the network (Han, Mao and Dally, 2016). It has been shown that parameters with the largest values make the most difference, so the parameters with the smallest magnitude can just be removed (Han et al., 2015). An example of pruning is shown in Figure 3.7.

0	0.89	0.67
-0.78	0	-0.99
0.63	0	0.53

Figure 3.7: Example pruned convolution kernel

SqueezeNet is one example of the application of both quantisation and pruning, where they were able to compress AlexNet to a 500× smaller size of 0.5MiB, whilst preserving the level of accuracy (Iandola et al., 2016). Note that whilst pruning can reduce model sizes by having all zeroes share a single memory location, it does not improve the inference speed.

Knowledge distillation is another compression technique. The primary idea of knowledge distillation is to train a smaller network, known as the student, to follow the same "thought proces" of a more complex network, known as the teacher (Buciluundefined, Caruana and Niculescu-Mizil, 2006; Hinton, Vinyals and Dean, 2015). The thought process of the teacher is encoded using soft targets, which are the predictions made by the teacher. Soft targets allow for ambiguity to be learnt, demonstrated by Figure 3.8.

$$\begin{array}{ccc} \begin{bmatrix} dog \\ cat \\ lion \\ fish \\ frog \end{bmatrix} & \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 0.121 \\ 0.735 \\ 0.083 \\ 0.031 \\ 0.030 \end{bmatrix} \\ \text{Labels} & \text{Hard Targets} & \text{Soft Targets} \end{array}$$

Figure 3.8: Hard vs soft targets in classification

This richer knowledge can be used to train networks more efficiently to match experts. Implementing knowledge distillation is simple, we train a teacher, and then train a student as normal. The only adjustment that needs to be made is that we not only compare the student's output with the target, we also compare its output with the teacher's output and linearly combine the two losses, shown in Equation 3.5.

$$\mathcal{L}_{overall} = (1 - \alpha)\mathcal{L}_{original} + \alpha\mathcal{L}_{teacher} \tag{3.5}$$

There have also been advancements to this knowledge distillation, such as ensemble, and quantised distillation. Ensemble distillation has multiple teacher networks, which can each be experts on a particular dataset or problem (Asif, Tang and Harrer, 2020). Quantised distillation focuses on the interaction between higher precision teachers and lower precision students (Shin, Boo and Sung, 2020). Interestingly, quantised distillation can allow students to share the same architecture as their teachers, but in a lower precision. We believe that there has been no work done to compare whether this is as effective as students having a smaller overall architecture.

For super-resolution, knowledge distillation has been extended to not only share knowledge at the final result, but also with intermediate feature maps (He et al., 2020). Binary super-resolution networks have been successfully enhanced with a similar type of distillation (Huang et al., 2021). EdgeSRGAN is an adaption of SRGAN trained for deployment on Edge TPU devices (Angarano et al., 2023). It makes use of transposed convolutions as opposed to the Pixel Shuffle operation because the authors claim it is more efficient, however this claim is yet to be quantified. EdgeSRGAN is also trained with the knowledge distillation, but the authors make no attempt to quantify the impact on the network with and without it.

The research to date has tended to focus on applying these techniques for generalised deep learning hardware, rather than making network-specific accelerators. The development of custom hardware is slow, even with high-level-synthesis, and making custom accelerators for every newly-trained architecture would take a very long time. Alongside this, hardware development is not often a skill that machine learning researchers are interested in pursuing. FINN is a recent project that aims to bridge this gap, by providing automatic compilation of trained neural networks to hardware descriptions (Umuroglu et al., 2017; Blott et al., 2018).

FINN implements a set of common neural network layers in high-level-synthesis. It then uses the compute graph of a neural network as an abstract syntax tree, allowing it to translate from the Quantised-Open Neural Network eXchange (QONNX) standard into those HLS layers. The HLS layers are then compiled into code in a hardware description language, and finally synthesised for the target architecture. The resulting hardware is static and not trainable; this is only done once we have a final model.

FINN is designed to be simple and accessible. We first train a neural network with quantisation. This can be done with a library such as Brevitas, which adds low-precision quantisation on top of PyTorch. Quantisation is essential for direct mapping to hardware, as simple integer adder circuits grow linearly with the number of bits, whereas multipliers grow quadratically (Abdelfattah, 2022a). Brevitas allows us to export our network to QONNX, and FINN will translate it to a custom representation, FINN-ONNX. This representation can be optimised by performing streamlining transformations, which eliminate floating point operations by moving them around and collapsing them (Umuroglu and Jahre, 2018).

Compiling networks with FINN is unfortunately not as easy as using a programming language compiler. It is more of a compiler infrastructure providing a software stack that interfaces with a compiler. Some transformations must be manually performed or baked into our compute graph, for example. We also have to provide FINN with a configuration file that specifies our target platform, and desired throughput, so that we can manually balance the throughput-area trade-off (Umuroglu et al., 2017; Blott et al., 2018).

Super-resolution accelerators have been made manually, but the work has been unable to compare the hardware size across different levels of quantisation (Lee et al., 2020). There has been work to develop super-resolution hardware with FINN, but they make no attempt of using deep compression techniques to reduce the bit-width (Su et al., 2024). FINN does not support pruned networks, however quantised distillation is a compression technique that preserves the model architecture (auphelia, 2023). Whether quantised distillation is useful for reducing the hardware size of these domain-specific architectures remains an open question.

Studies have been done to compare the performance of networks that have been mapped to hardware through FINN, and deployed on a GPU. One particular study favours FINN, but deploys a binary neural network on the FPGA, and then a full precision network on the GPU

(Pettersson, 2020). Another also favours FINN, but compares a custom development board with a £260 FPGA alone, with a £100 NVIDIA Jetson Nano computer (Hamanaka et al., 2023; NVIDIA, n.d.; AMD, n.d.c). We believe no fair study has been done comparing the performance per unit cost of GPU implementations and direct hardware mapping.

HLS4ML is an alternative to FINN with the same goal of mapping neural networks directly to hardware (Duarte et al., 2018). HLS4ML has full support for QKeras, a quantisation library build on top of TensorFlow, but has limited support for PyTorch and the QONNX standard (au2 et al., 2021; Ngadiuba et al., 2021; Aarrestad et al., 2021; Ghielmetti et al., 2022). This will likely change in the future, as the QONNX standard is a collaboration project between the developers of FINN and HLS4ML. The main difference between them is that FINN is tailored for large low-precision networks, whereas HLS4ML prefers small but higher-precision networks.

When it comes to the cost of ASIC development for companies, the primary costs are in labour (Hennessy, Patterson and Asanović, 2012). Figure 3.9 shows a breakdown of ASIC development costs, however the information is from 2011, and may not reflect modern development. It also comes from only surveying others. It would be useful for companies to know the cost of developing ASICs that implement image-generating networks, and whether tools such as FINN can help reduce that overall cost.

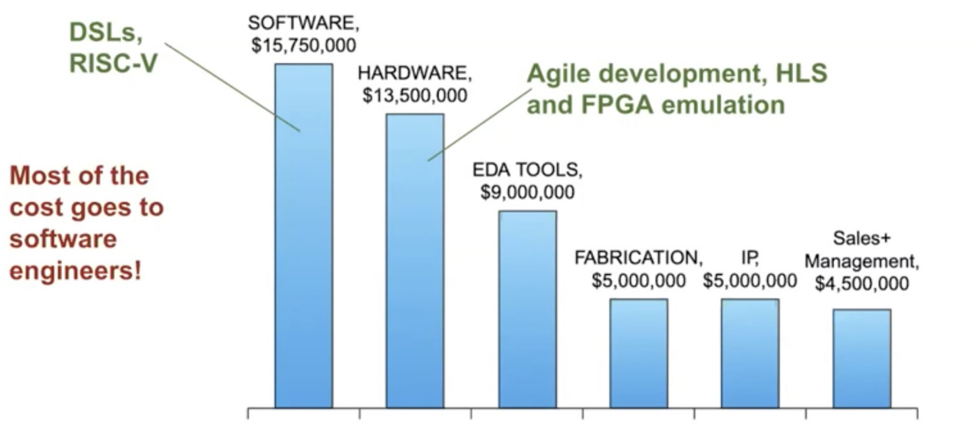


Figure 7.51 The breakdown of the $50 million cost of a custom ASIC that came from surveying others (Olofsson, 2011). The author wrote that his company spent just $2 million for its ASIC.

Figure 3.9: Development costs of a custom ASIC
(Hennessy, Patterson and Asanović, 2012; Abdelfattah, 2022b)

Having reviewed the state-of-the-art in this area, we refined our research question to more accurately reflect present knowledge gaps. We saw that whilst image-generating models can be mapped directly to hardware, it is unknown how that hardware compares to consumer GPUs in terms of performance and production costs. It is also unclear whether the impact of quantised distillation is great enough to reduce the bit-width of a quantised neural network without a loss in quality. If this was the case, quantised distillation could be an effective strategy for producing super-resolution ASICs at lower costs. We now pose our refined research question:

Does quantised distillation allow for cheaper image-generation hardware?

Chapter 4

Methodology

To answer our refined research question we proposed a series of hypotheses and experiments. Primarily, we were interested in seeing if quantised distillation could be used to reduce the bit-width of a model without seeing a loss in generation quality. We have seen that it is unclear whether quantised distillation is more effective when the student has a different architecture to its teacher, or when it has the same. This is important for our study as we may be able to not only reduce the bit-width of our networks, but also the overall parameter count.

Furthermore, we were focused on developing a methodology that would allow us to compare the economics of mapping neural networks to hardware, with performing inference on a GPU. We therefore posed the following hypotheses to investigate:

1. Quantised distillation can reduce model bit-width without raising the LPIPS score
2. Mapping neural networks directly to hardware results in greater inference throughput per unit cost than consumer GPUs

We denote our hypotheses as h_1 and h_2. The first hypothesis allows us to explicitly quantify the impact of quantised distillation. This can be measured through the percentage change in LPIPS scores. The second hypothesis allows us to compare the economics, considering the commercial validity of having a dedicated super-resolution accelerator.

4.1 Experiment Design

For testing our hypothesis we conducted a series of experiments. We measured the LPIPS scores of quantised networks without knowledge distillation, and the percentage change in LPIPS they experienced with knowledge distillation. We then used the FINN compiler on our trained network, and measured the expected LUT count for particular throughputs. It is important to note that we are not targeting state-of-the-art super-resolution quality.

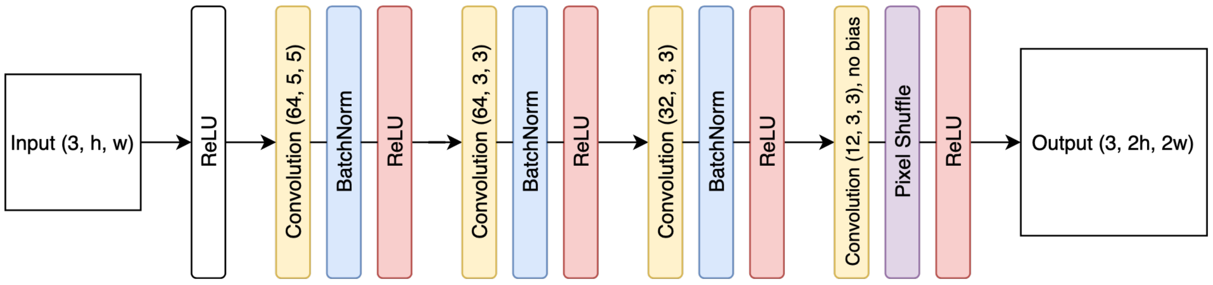
Before our main experiments, it was worth investigating some of the questions we raised during our literature review, as they could impact our visual quality and measurements. Notably:

1. Does the $\mathcal{L}_1$ loss outperform the $\mathcal{L}_2$ loss for training quantised super-resolution networks?
2. How much faster is a transposed convolutional layer than a Pixel Shuffle operation?
3. Does student-architecture affect the impact of quantised distillation?

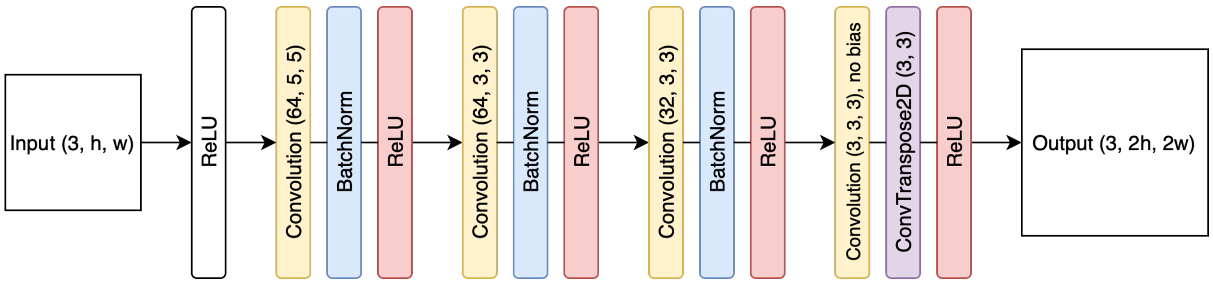
The summary of the experiments we undertook is as follows:

1. Measuring the convergence of the $\mathcal{L}_1$ and $\mathcal{L}_2$ losses for training
2. Measuring the throughput of the transposed convolutional and Pixel Shuffle layers
3. Measuring the change in LPIPS resulting from quantised distillation
4. Measuring the cost of the hardware generated by FINN in comparison to GPUs

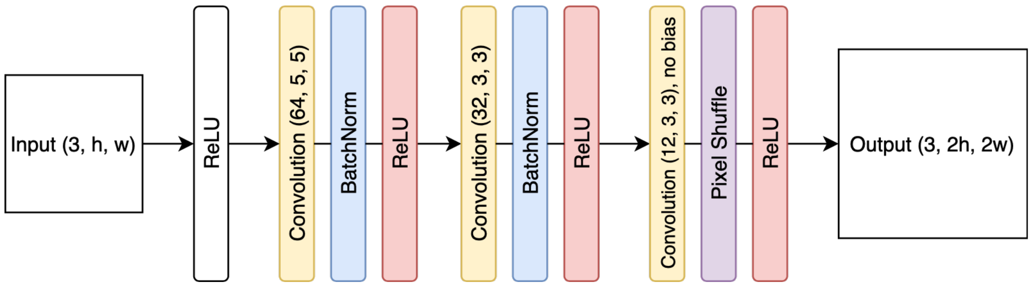
Throughout our experiments we used the previously implemented quantised version of the ESPCN network (Colbert, Pappalardo and Petri-Koenig, 2023). This network is a simple super-resolution network featuring the pixel shuffle layer for upsampling. We chose this network as it is fully-convolutional, and is the foundation for all of the advanced models of modern research; the results we find should carry over to those models. We also developed two variants of the model, a Faster-ESPCN (FESPCN), which uses a transposed convolutional layer instead of the pixel shuffle layer, and a Mini-ESPCN (MESPCN), which removes one of the convolutional layers. Each architecture is shown in Figure 4.1.



(a) ESPCN



(b) FESPCN



(c) MESPCN

Figure 4.1: Network architectures

We followed a similar training approach to the training of the original quantised ESPCN, but with some differences to save on compute time (Colbert, Pappalardo and Petri-Koenig, 2023). We split the high-quality DIV2K dataset into a training set and validation set, using the validation set to adjust hyperparameters. The only hyperparameter we were really interested in testing was the loss function for our first experiment. For every network we trained, we repeated it three times to calculate the confidence of our results.

For our first experiment we trained ESPCN six times, three using each loss function. We compared their performance on the validation set, and carried the better performing loss function forward for future experiments. For the remaining experiments, we needed to train a series of baseline models. We trained each of our architectures on five levels of precision, FP32, INT8, INT6, INT4, and INT2, providing a broad range of quantisation strength. Integer precisions were chosen in particular as we have seen they would reduce our final circuit size. We then measured their performance on the commonly used BSD100, Urban100, Set5, and Set14 test sets for comparability with other work (Agustsson and Timofte, 2017; Martin et al., 2001; Bevilacqua et al., 2012; Zeyde, Elad and Protter, 2012; Huang, Singh and Ahuja, 2015).

For each training image, we sample a random 512×512 px patch, instead of the full image, to save on compute. This is because we are more interested in the *preserved* generation quality rather than the generation quality itself. As we are training on less data per epoch than the original QESPCN, we also decay our gradient descent step size less aggressively, using the Adam optimiser. Finally, we train for 200 epochs instead of 100. Whilst we could have increased our patch size, 512×512px was optimal for maximising our GPU throughput. All of our training was done using an NVIDIA RTX 3080 Laptop GPU (details in Appendix A.3).

Picking random patches allowed the networks to train on the detailed parts of images. If we chose to sample only the corners, for example, we would be training our networks on lower-frequency patches, such as the sky in Figure 4.2. High-frequency patches are harder to upscale, and should be the focus for deep-learning approaches.

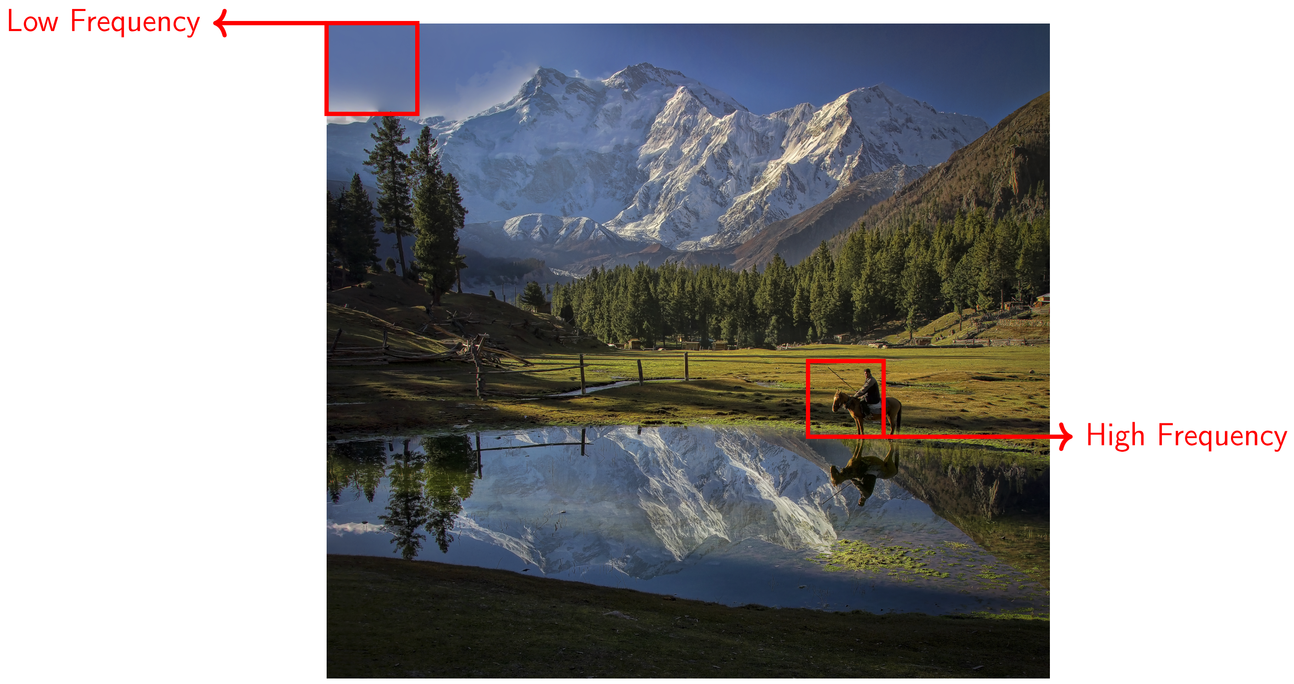


Figure 4.2: Lower vs higher frequency image patches (Agustsson and Timofte, 2017)

For the quantisation technique, we chose to perform asymmetric quantisation on the activation functions and the trainable weights. The accumulators were not quantised and were each INT32, as they require a larger range. We also chose to use linear quantisation as non-linear quantisation does not help in reducing the hardware size. The quantisation was done per-layer, as being done globally would reduce performance without reducing the overall hardware size. We also performed quantisation during training, increasing our visual quality and consistency at the cost of longer training times.

Having trained our baseline models, we now needed to incorporate quantised distillation. We focused on two primary types, allowing us to answer question 3. The first was distilling knowledge from the INT8 ESPCN network to the INT6, INT4, and INT2 ESPCN networks, which we refer to as same-architecture distillation. Secondly, we distilled knowledge from the INT8 ESPCN network to the INT8, INT6, INT4, and INT2 Mini-ESPCN networks, which we refer to as different-architecture distillation.

We used knowledge distillation only at the output of our networks, as opposed to the comparing feature maps throughout the networks, as in previous work (He et al., 2020). This was because our networks are relatively small, and all of the feature maps are closely related to the final output image. Implementing this type of distillation also allowed us to save on compute.

For each possible configuration of knowledge distillation, we also varied the hyperparameter α, which controls how much the student listens to the soft targets from the teacher instead of using the hard targets. We picked four values of α: 0.05, 0.2, 0.3, and 0.8. This range was selected as it covers prioritising both hard and soft targets at different intensities.

Our experiments and hypothesis focus on the LPIPS score for evaluating image similarity, as we have seen it is a more reliable metric than the PSNR and SSIM metrics. We did not consider using it as a loss function, as it would perform the forward pass of another network for each sample, drastically increasing our training times.

Finally, for creating the hardware designs we used the FINN compiler. To investigate our second hypothesis fairly, we first measured the maximal throughput we could attain on our GPU. We then used this measured throughput on each network architecture as our target throughput for the FINN compiler. FINN would then generate the resulting hardware and we would measure its size in LUTs. We discarded the BRAM as the type of architectures FINN produces (streaming dataflow architectures) do not use much memory.

We first had to measure our GPU throughput, which can be a difficult task due to their complexity. We followed advice from experts in this area, using a modified version of their code tailored for our measurements (Deci, 2023). This code primarily ensured that the GPU was in an active state, and that we were not measuring data transfer overheads, only operating on the GPU. We also used the integrated API for measurements from our deep learning library.

The FINN compiler cannot produce an exact throughput target, so we measured the size of the hardware that produced the most similar throughput as our GPU. We discuss how fair this is, along with other assumptions made, in our evaluation. Having determined the size of the hardware, we then found the price of the cheapest FPGA development board (system-on-chip) that could load that hardware. We chose the development board, instead of the FPGA alone, as graphics cards feature PCIe connectors (interfaces to motherboards) that a real super-resolution ASIC would have to add in comparison to an FPGA prototype.

Whilst we would have liked to measure the power efficiency of the hardware we create, this is impossible as the boards that we have available can only measure the power usage of the whole system-on-chip. This is because there is a shared rail between the programmable logic and the processor, which supplies the electricity (McCabe, 2019).

4.2 Implementation

To implement our neural network, we used the PyTorch and Brevitas frameworks. PyTorch is a general compute-graph framework that has support for neural networks, and is generally the preferred tool among machine learning researchers. Brevitas is a library built on top of PyTorch, that includes support for linear, asymmetric quantisation. Networks are designed in exactly the same way, but use building blocks that support quantisation, such as a `QuantConv2d` layer as opposed to a `Conv2d` layer.

Brevitas implements the fake-quantisation method to keep our network differentiable. That is, it will apply quantisation dynamically during the forward-pass; we were performing quantisation-aware training. Brevitas supports the export of our graph to QONNX, which is the compatible interface for the FINN compiler. Both Brevitas and FINN are open-sourced, and maintained by Xilinx, so they are very compatible with one another, and using them together is more fluid than alternatives.

The FINN compiler infrastructure is a large project with complex dependencies, and is not available on all systems. To use FINN we had to prepare a virtual machine with Ubuntu 18.04, over 400GiB of disk space, and over 16GiB of dedicated RAM. Even with these resources, FINN will run only inside a Docker container (lightweight equivalent of a virtual machine) that was broken at the time of development. We found a solution online which enabled us to continue with the project (Daiphys, 2023).

Our codebase is a standard codebase for the training of a machine learning model. We have six small python packages organised into utilities, training, neural networks, metrics, exporting, and a package for CUDA support. We also have a directory of scripts that were used throughout for testing properties and performing operations on datasets. There is finally the main two scripts, `main.py` and `main_kd.py`, which start the training process with or without knowledge distillation. For efficient training, there were also Windows batch scripts (`run.bat`, `run_kd.bat`) which would iterate over different configurations, train and test them, and export all the required data without any human involvement. The full codebase can be found at the link in Appendix E, or in `jake-davies-code.zip`.

To implement Mini-ESPCN and Faster-ESPCN, we modify the models provided in Brevitas through inheritance and overriding their structure (Colbert, Pappalardo and Petri-Koenig, 2023). Knowledge distillation was the only feature that required any changing to the training loop. Before we perform backpropagation, we check to see if there is a teacher model, and if there is then we linearly combine the losses, as in Equation 3.5. One interesting problem that came about when using knowledge distillation was fitting both networks in GPU memory.

Whilst we could have simply decreased the batch size for these experiments, this would have made our comparisons unfair (as batch size affects training). Instead, we adopted the gradient accumulation technique often used for training large language models (pseudo-code in Appendix A.2) (Raschka, 2023). To implement gradient accumulation, we reduced the batch size by a factor of n, and accumulated the losses. We then only perform the backpropagation step every *n* batches, resulting in the same result as if we had used a single, larger batch, whilst reducing GPU memory load.

Caching was an essential feature that drastically improved training times. Python provides the `@cache` function annotation which uses the function arguments as keys to a hash map, which maps arguments to previous results. We applied this annotation to our function which loads images from disk, and were able to barely store the whole dataset within 32GiB of main memory. Through this annotation, we will able to improve training times from over two minutes per epoch, to under ten seconds per epoch. This 12$\times$ speedup often is glossed over in the research papers, but is a crucial implementation detail.

The graphics card we used for training was a laptop graphics card; it is not sold separately (Appendix A.3). To fairly compare the prices of such a card, we found that the RTX 3060-Ti graphics card performs extremely similarly, within 3% effective speed (UserBenchmark, 2024).

To generate each design for FPGA deployment, we had to write some code to interact with the FINN compiler (see `compile.py`). This comes from the fact that whilst FINN has a compiler, we need to apply transformations ourselves which suit our neural network architecture. Our transformations take heavy inspiration from the standard digit classification CNN example provided by the FINN team (Umuroglu et al., 2017; Blott et al., 2018). We remove transformations related to linear layers, or classification. Importantly, we bake in an image transformation step that divides the RGB values by 255, as whilst PyTorch does this for us, FINN is not aware of it.

We were able to test the functionality of our network at various points in time. The first was during training, where we measured the similarity score each epoch to ensure our network was learning. We also wrote an example image file from the network at the end of each epoch to verify that it was learning a super-resolution function, rather than noise.

The next place we tested was after the export to the QONNX format, where we ensure the loaded model runs correctly on that same example image we gave it. Finally, after applying each streamlining transformation, we test the remaining FINN-ONNX network. This test was important as it is what the FINN compiler uses in order to generate each design.

Overall, we trained 159 neural networks for over 97 compute hours, resulting in a collection of over 31,800 generated images. Samples are shown in Appendix C. The full set of post-training images for each model can be found along with this upload in `jake-davies-images.zip`. The full set of trained model weights, and QONNX exports can be found in `jake-davies-models.zip`.

Chapter 5

Results and Evaluation

Having described our methodology, experiments, and implementation, we now present the evaluation of our results. We begin by determining the impact that quantised distillation had on both our student architectures across varying levels of precision. We later discuss the economics of our created hardware and implications for the accelerator market. Along the way we critique our approach, and discuss the validity of any assumptions we have made.

Some visualisations in this chapter plot the standard error associated with running each experiment three times. Plots that calculate new results combine the errors using overestimates to ensure that we are not making unjustified claims. Colours were chosen by the `distinctipy` library for visual clarity and colourblind-accessibility (Roberts et al., 2024).

Our analysis is directed by the following questions: How much impact does quantised distillation have on our networks? How does this differ between same-architecture and different-architecture students? How do our results change as the bit width decreases? What is the impact of varying the parameter α? Does FPGA throughput outperform GPU throughput per unit cost?

5.1 Visual Quality

The very first experiment we ran was to determine the loss function to use for the remainder of the experiments. We clearly show that for our networks and training regime, the $\mathcal{L}_2$ loss performed better. This goes against what other work has found for super-resolution, however we are not surprised (Zhao et al., 2018). Unlike previous work, our network is quantised, quite small, and we do not train on the full dataset – but random patches. Our results were ran three times and consistent. The plots for this experiment can be found in Appendix B.1.

We first trained our networks without any knowledge distillation, with results during training visualised in Appendix B.2. Our post-training test set performance for each baseline architecture is shown in Figure 5.1.

The results on our test sets made it very clear. For the quantity of data we trained on, the performance of FESPCN was very inconsistent across runs, yet it always performed the worst. Due to such a large margin of error during training, we chose to not use this architecture for future experiments. It also produced checkerboard artefacts, shown in Figure 5.2. However, we sought to investigate whether it demonstrated higher inference throughput than the Pixel Shuffle operation. In our measurements, it was in fact the slowest (Appendix B.6).

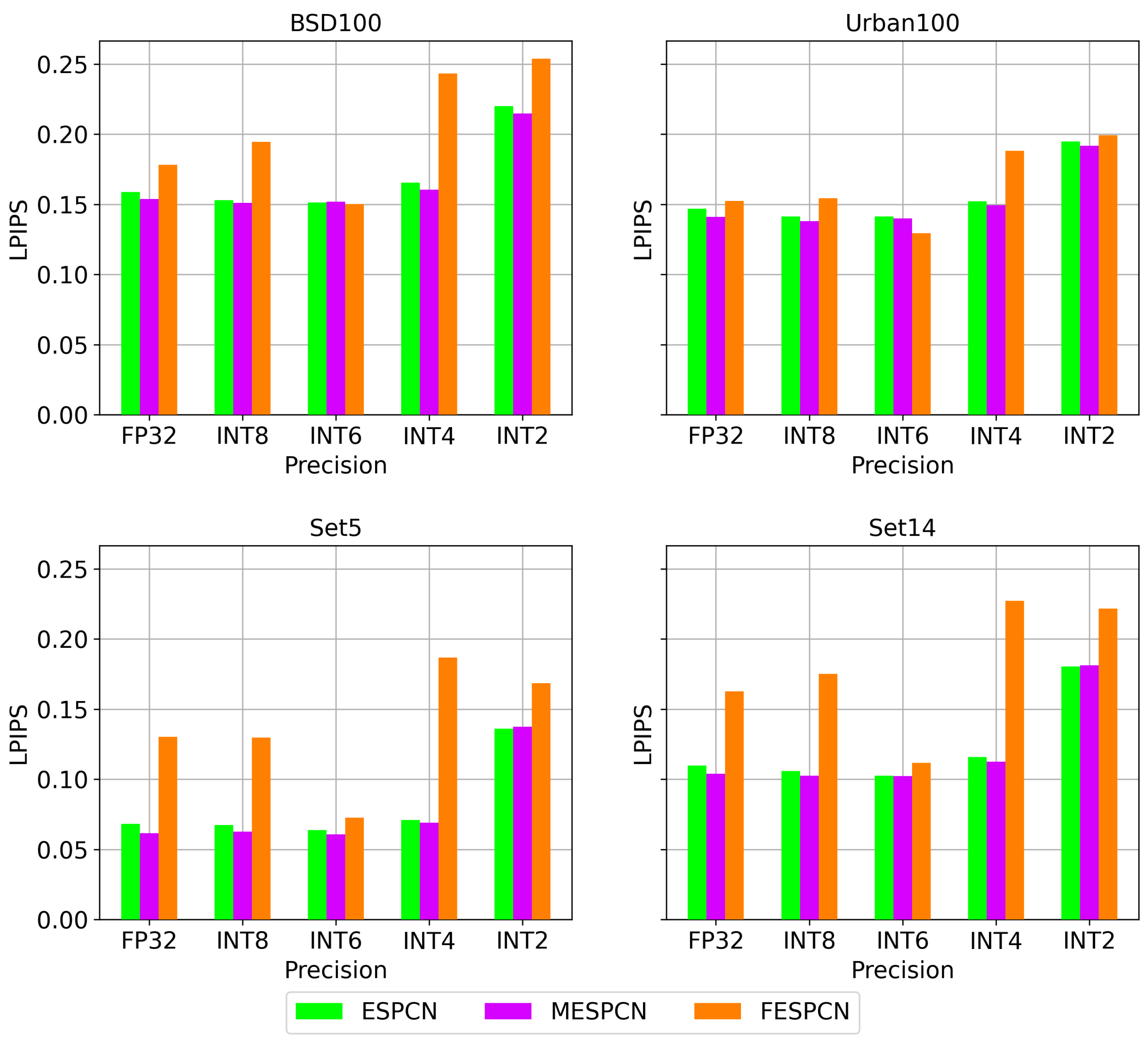
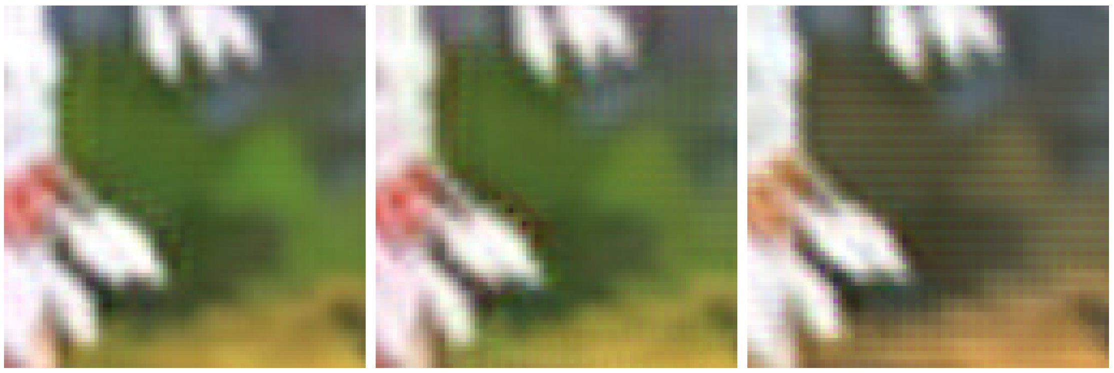


Figure 5.1: Average LPIPS scores across test sets (lower is better)



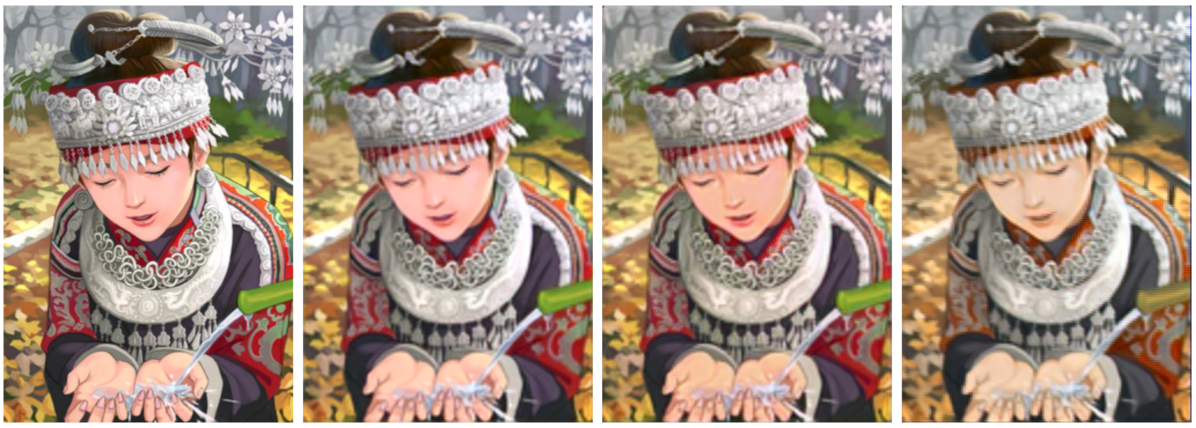
(a) ESPCN (b) MESPCN (c) FESPCN

Figure 5.2: Example renders (zoomed from Figure 5.3)

Something interesting to note is that our smaller network, MESPCN, outperformed our larger ESPCN network a large majority of the time. There were only 3 out of 20 situations where it was barely outperformed, being INT6 precision on BSD100, and INT2 on Set5 and Set14.

These results were minor but consistent, and the error alone was not enough to disregard them. Figure 5.2 shows the ESPCN network producing accurate results, with the MESPCN render producing incorrect black pixels. Figure 5.3 shows the MESPCN producing harsher noise, with a colour tone that does not match the ground truth image. Overall, the ESPCN network produces images that we perceive as more visually correct, despite having a worse LPIPS score.

This has implications for the interpretation of our remaining results. The first implication is that whilst LPIPS is an improvement over previous metrics, it is still not a perfect metric for perceptual quality. The second is that we have distilled knowledge from a teacher network that statistically performs worse but visually performs better. This makes measuring the impact of quantised distillation difficult to analyse.



(a) Ground truth (b) ESPCN (c) MESPCN (d) FESPCN

Figure 5.3: Example images produced from INT8 networks

Having ran our initial experiments, we had to decide on which network would be a suitable teacher. The choice of the highest-precision ESPCN network seems obvious, however when we visualise the images it produced, we see that it produced some strong artefacts (Figure 5.4). These artefacts were likely caused by having too large a step size during gradient descent, leading it to never converge. Due to this, we proceeded with using the INT8 version of our network as the teacher. The outputs for relevant networks are visualised in Appendix C.

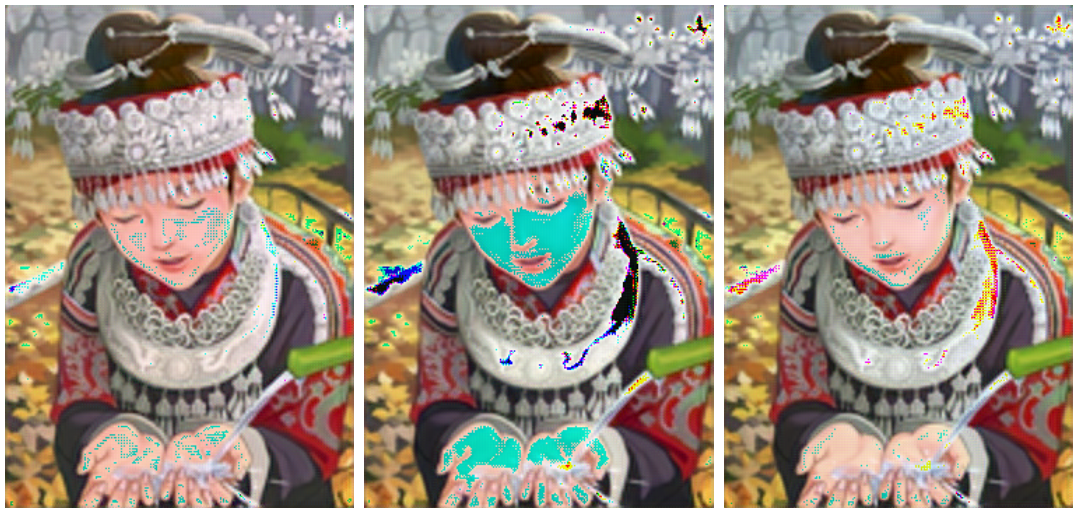


Figure 5.4: Example images produced from FP32 ESPCN

Our primary experiment was to compare same-architecture and different-architecture quantised distillation. As we have found the LPIPS metric to be somewhat misleading, we not only compared the difference in LPIPS measured, but also how we perceived the generated results. The results for INT2 networks are shown in Figure 5.5, where green bars represent ESPCN (same-architecture distillation), and purple bars represent MESPCN (different-architecture distillation). Lower scores are better. The impact of quantised distillation during training is visualised in Appendix B.4. The same data for other bit widths is visualised in Appendix B.5.

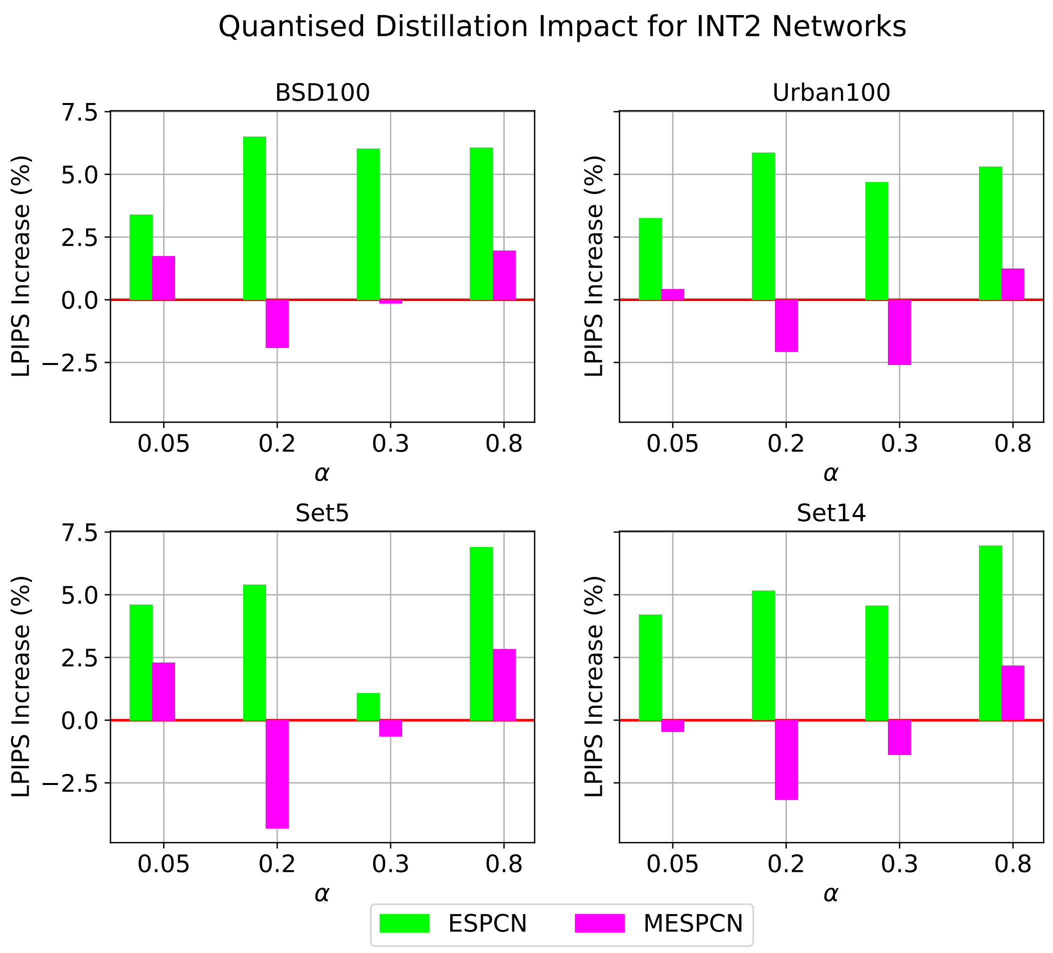


Figure 5.5: Increase in LPIPS scores (INT2) (lower is better)

We want the plotted lines to be below the red lines, as this represents where our LPIPS score has decreased. The full results for each bit width demonstrate that the 2-bit networks were the only networks to receive better LPIPS scores on our test sets, with the best result being a 4.3% decrease in LPIPS. The 4-bit, 6-bit, and 8-bit networks reacted badly to the quantised distillation and received almost a 25% increase in LPIPS score in the worst case.

The worst case was using $\alpha = 0.2$ for same-architecture INT4 networks. We believe that for networks of higher precision than INT2, having the basic pixel-wise knowledge distillation augmenting the loss resulted in noise being added during training. This could explain why not only were most networks worse off with knowledge distillation, but they also performed worst during the training process, visualised in Appendix B.4. Overall, we do not see any evidence that quantised distillation can reduce the bit-width of our architecture without it resulting in a loss in quality; we reject our first hypothesis.

This noise would take shape in combining the two losses before gradient descent. We believe that the INT8 teacher was not producing accurate enough results to improve the gradients computed for gradient descent. The images that the INT8 ESPCN network produced appear to be the ground truth image but with a small amount of Gaussian noise applied to them (Figure 5.3, Appendix Figure C.2). This means our teacher was encouraging the production of noisy images, which caused students to produce images less-alike to the ground truth targets.

In the case of INT2 networks, we did see an improvement. We now believe it is only because they produced extremely poor results to begin with (Appendix Figure C.5). When we compare the results of two-bit networks with and without quantised distillation in Figure 5.6, we do not see a major difference, but we do see sharper objects (the fence and the green spout).

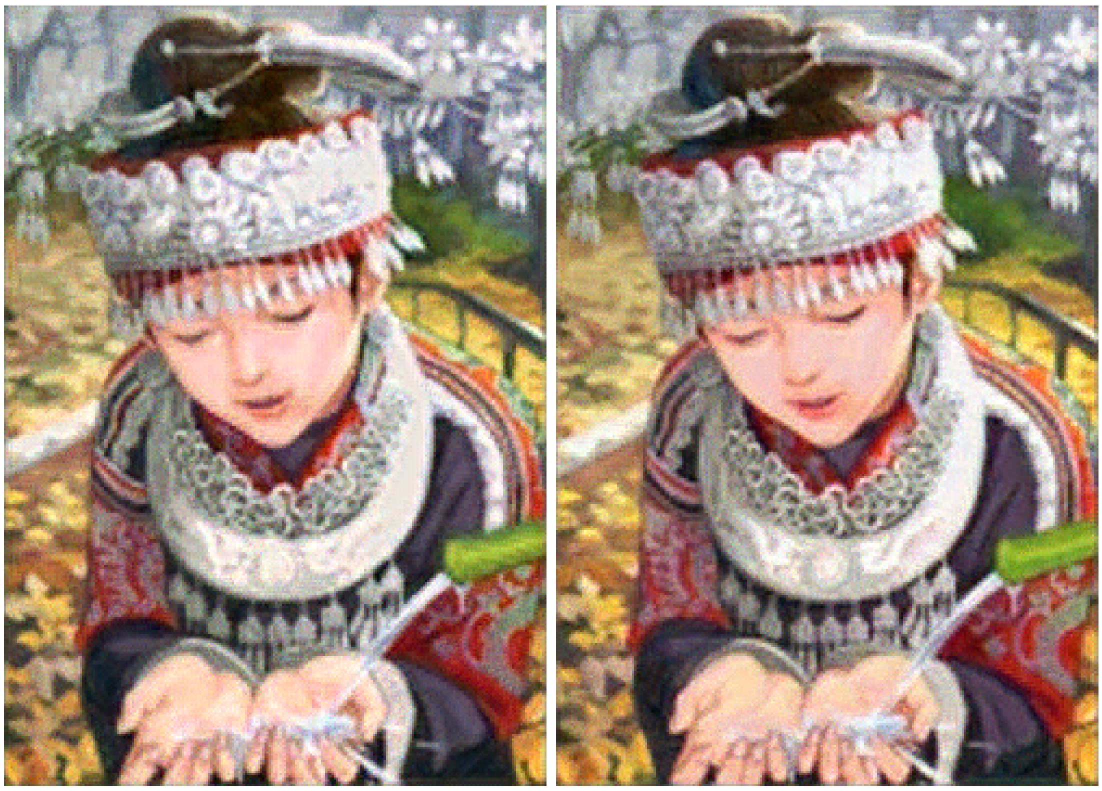

(a) Without quantised distillation (b) With quantised distillation

Figure 5.6: Results produced by INT2 MESPCN networks

For our networks, the different-architecture student had a far more positive impact from knowledge distillation. This was consistent; there was not a single case where the same-architecture student saw more benefit from quantised distillation, across various choices of α and network bit-width. We are unsure why this is the case, but believe it could be because the smaller student architectures are forced to generalise. This is because they have less parameters and cannot mimic their teachers exactly, whereas for the larger architectures it is possible with only a slight reduction in precision.

This series of networks gained the most benefit when the teacher had less involvement in their learning, with an optimal value for α being around 0.2. This is likely because our networks are already relatively small in comparison to the state-of-the-art. We are also only using a simple form of quantised distillation – we are comparing outputs pixel-wise. It appears that for the teacher to be more useful, it should be far more advanced itself, and not only higher-precision. We can make an analogy to undergraduate students, who prefer learning from experienced lecturers as opposed to graduate teaching assistants (Park, 2002). It would be interesting for further work to explore extreme quantised distillation, where the teacher network is reminiscent of GigaGAN or the Efficient Super-Resolution Transformer, and the student is a tiny 4-bit network alike MESPCN. These teacher networks likely contain only a few significant feature maps that a tiny network may learn to imitate.

The claim that the teacher should be far more advanced could also be false; our data is limited. Whilst we had to run each experiment at least three times to gather consistent results, this limited our ability for the mass collection of data. We could be observing a local minima for α, when the truly optimal value could be 0.9, for example.

One assumption that we implicitly made through our reduced training regime, is that quantised distillation would not only improve our final results, but converge faster. Since our models are not necessarily trained to convergence, we should be careful making claims about final performance. Whilst it looks like our networks have converged in Appendix B.2, it is unclear whether LPIPS scales logarithmically, meaning there could be far more reconstruction quality to gain. However, we believe that our networks had mostly converged and would experience little benefit from further training. The performance at various epochs is visualised in Figure 5.7 and Figure 5.8 to support this claim.

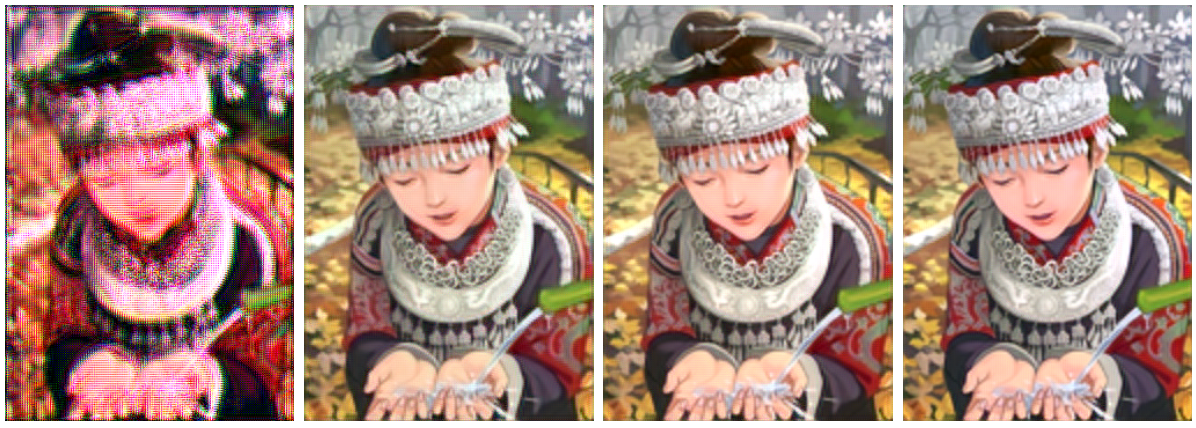

(a) Epoch 1 (b) Epoch 50 (c) Epoch 100 (d) Epoch 200

Figure 5.7: Results produced by INT8 ESPCN

One final critique we can make on the evaluation of our visual quality, is that we have primarily focused our perceptual quality on this single output image. It is completely possible for this image in particular to be an outlier, not representing the final distribution. However, producing over 200 outputs for various images during a training run was infeasible for our available storage. For a more thorough evaluation, we would expect many, diverse images to be analysed and compared. We believe our evaluation is likely to be appropriate, as this image does not appear in our model's training set; it is a blind test.

The discovery of the impact of student-architectures is useful for other researchers exploring compression via quantised distillation. They should ensure that not only do their teachers have higher precisions, but also be more advanced networks that generate far higher-quality outputs. Having said this, they should also experiment with the more advanced pixel-correlation loss for intermediate feature map distillation, which may produce different results (He et al., 2020).

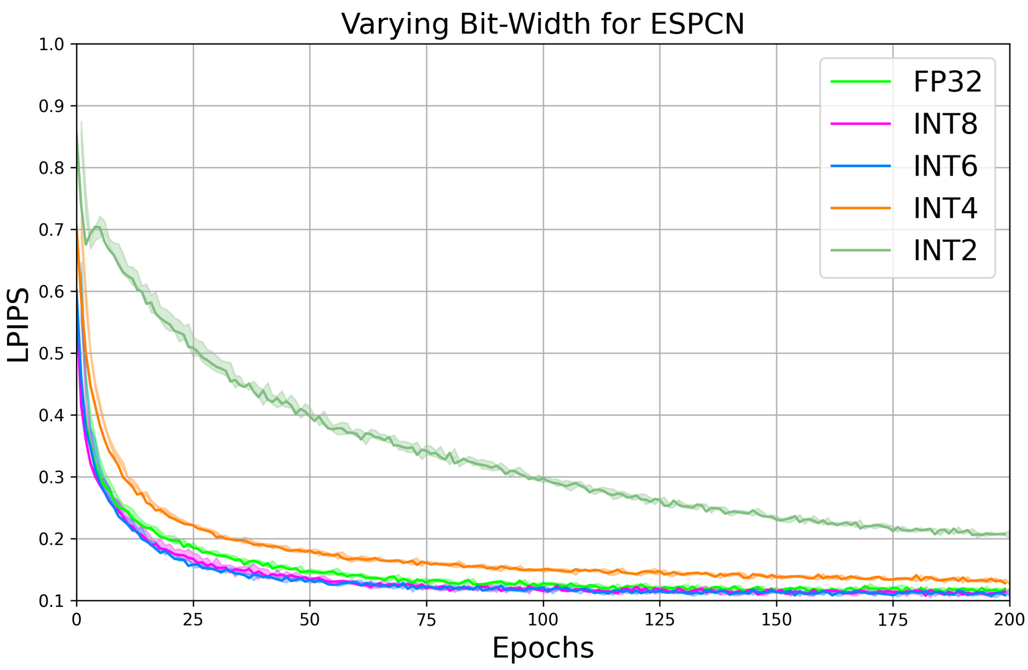


Figure 5.8: Validation set performance during training for INT8 ESPCN

5.2 Economics

Our second hypothesis required us to measure the throughput of our networks deployed on a GPU, and mapped directly to hardware. This section aims to answer whether or not super-resolution accelerators are more economically viable than graphics cards for consumers. We also evaluate if quantised distillation has proven to be an effective technique for reducing the potential cost of such a chip. By nature, there will be more assumptions and approximations made in this section than the previous, as we have not produced a full, standalone super-resolution ASIC, but a prototype deployable on an FPGA through the FINN compiler.

We ran each network with the maximal batch size for our graphics card (Appendix A.3) post-training. The best performing network for throughput was the FP32 network, which makes sense as our graphics card natively supports the FP32 datatype (TechPowerUp, 2024). The first assumption that we made, is that it is fair to compare the FP32 throughput of a graphics card with the INT8 performance of our dedicated accelerator. As shown in Table 5.1, these precisions achieved very similar visual similarity.

	ESPCN	**MESPCN**	**FESPCN**
BSD100	0.1587	0.1537	0.1782
Urban100	0.1469	0.1411	0.1524
Set5	0.0680	0.0616	0.1302
Set14	0.1098	0.1040	0.1627

FP32 Networks

	ESPCN	**MESPCN**	**FESPCN**
BSD100	0.1529	0.1511	0.1944
Urban100	0.1413	0.1379	0.1542
Set5	0.0674	0.0627	0.1296
Set14	0.1058	0.1024	0.1751

INT8 Networks

Table 5.1: Test set evaluation (average LPIPS scores)

It could be argued that we should not use the floating point networks; they produced artefacts.

However, we have seen successful implementations using the this representation before, and an adjustment to our training could likely fix this issue (Shi et al., 2016).

Having found our GPU throughput, we then compiled every network architecture into a hardware description using the FINN compiler. The FINN compiler allows us to set a desired throughput, and generates designs accordingly. The FINN compiler primarily relies on duplicating hardware that can run in parallel to increase throughput. As such, the compiler cannot generate designs at arbitrary throughput; an approximation had to be made here. This was that we take the closest possible throughput to our GPU, and measure that designs complexity. We believe this is reasonable because the approximations are relatively close to their targets. The results of lookup table usage are shown in Table 5.2.

Datatype	ESPCN	MESPCN	FESPCN
INT8	2,606,884	1,751,936	2,606,920
INT6	1,543,649	1,219,405	2,447,761
INT4	1,129,185	1,011,277	1,129,185
INT2	737,839	939,118	690,259

Table 5.2: Lookup table usage per architecture (6-bit-input)

This hardware was generated by considering designs with less than our target throughput. For ESPCN and FESPCN, the throughput our GPU achieved was about 140 fps for a 512×512px image. Mapped onto hardware, these architectures achieved a 127 fps throughput, and the alternative was 190 fps. For MESPCN, our graphics card achieved 240 fps, whereas our custom hardware achieved only 190 fps. This is the greater approximation, however the alternative was a drastic 375 fps.

This table shows us two notable results. The first is that the ESPCN and FESPCN architectures were extremely similar in lookup table usage when compiled to hardware. This makes sense as their overall parameter count is roughly the same. The second is that whilst MESPCN was much smaller overall, the reduction in size it saw from various bit widths was smaller than the other networks. This suggests that a lot of the lookup tables are used for overhead.

We make another assumption to bound our predicted costs, and claim that an ASIC would cost a similar amount to a full FPGA board (system-on-chip) that was capable of loading the generated design onto it. This is because to be comparable to a GPU, this ASIC would need to translate from the AXI streaming interface (the interface FINN can compile to), to the PCIe interface, which is required for support with consumer motherboards. GPUs are also sold with only a select few choices of memory size, limiting their yield, therefore increasing their price when fabricated onto silicon.

Having searched for suitable boards, Table 5.3 summarises our findings. Each board is often marketed in number of "Logic Cells", each of which (currently) represents approximately 0.46 6-bit input lookup tables (avrumw, 2021).

Board Name	Price	LUTs (approx.)
PYNQ-Z2 (AMD, n.d.b)	£103	53,200
Mercury+ XU7 Zynq UltraScale+ (AMD, n.d.d)	£1,600	276,000
Virtex UltraScale+ VCU118 (AMD, n.d.a)	£7,223	1,163,700
Virtex UltraScale+ VCU129 (AMD, n.d.e)	£15,484	1,701,700

Table 5.3: FPGA boards

Quantised distillation did not prove useful for reducing the overall size or cost of our hardware. In our results, the best change was the 2-bit MESPCN which saw a 4.3% decrease in its LPIPS score. The change was unfortunately not enough to make it comparable to an 4-bit implementation, which would have allowed us to save 7% of our total lookup table use.

If INT6 to INT4 quantised distillation was effective, one would be able to save over £8,000 switching from the Virtex UltraScale+ VCU129 to VCU118. Whilst this would be impressive, it is unfortunately not the outcome we observed. We found that for our GPU configuration, we can find an equivalent GPU costing under £400 (UserBenchmark, 2024). It is clear that we cannot get anywhere close to the price of GPUs when mapping even relatively small neural networks directly to hardware. The number of lookup tables used scales poorly as the network gets even deeper (Umuroglu et al., 2017; Blott et al., 2018). With this evidence, we must reject also our second hypothesis, and state that with the current state of mapping, custom hardware would be far too expensive to compete at the consumer level.

Something to consider is whether or not it is reasonable for deep learning hardware to be automatically generated if it were to be produced. We believe that in the future it could be, for numerous reasons. The first is that as computer science and machine learning separate as disciplines, there is an ever-growing skills gap between machine learning researchers, and hardware engineers. As networks get more and more comlex, this skills gap would become more of a problem during development. The second is that we have already seen a huge cost of ASIC development itself is the cost of the engineers (Figure 3.9). Automatic compilation could allow for significant savings to be made for businesses in the industry. Finally, the development time of such large and complicated hardware would be enormous. Previous work has never made multiple of these accelerators manually, whereas we were able to create 12 relatively quickly through FINN. The resulting hardware size is clearly the main barrier for this hardware.

The usability of this hardware is also a major flaw. FINN primarily uses a compute graph for its mapping, but the graph must come with a fixed input node. This means that whilst on the GPU our network could perform inference on images of any size, the generated hardware is fixed to a specific shape of tensor. This is a significant problem for general use. We believe that whilst not so useful for desktops themselves, these chips may eventually find a place in consumer laptops, monitors, and TVs, if they were ever to be produced.

Chapter 6

Conclusions

Having presented our findings and analysis, we now turn back to our original research question.

Does quantised distillation allow for cheaper image-generation hardware?

Having considered our results and hypothesis, our answer is evident. For our networks and implementation, quantised distillation was unable to reduce the cost of the produced hardware. However, we acknowledge that many simplifications and approximations were made in our methodology. We still believe that with more capable teacher networks, along with more advanced distillation techniques, quantised distillation may prove to be an effective technique for this in the future. Furthermore, our economic analysis was approximate, however we do believe it is useful for businesses attempting to estimate the costs involved in this area.

Despite not achieving desirable results, we were still able to answer our research question reliably whilst making notable discoveries in the process. We believe our evaluation on the impact of student architectures in quantised distillation is a novel and useful contribution to deep compression techniques. Additionally, we were able to disprove that the transposed convolutional layer is less computationally demanding than the Pixel Shuffle operation.

There is still a lot of potential work that can be explored in this area; we pose some inspiration for future work that we were not able to fully explore. Whilst it has been shown that the $\mathcal{L}_1$ loss should perform better than the $\mathcal{L}_2$ loss for training super-resolution networks, we found the opposite for our quantised network. It would be worth investigating how quantisation impacts hyperparameter selection for training super-resolution networks.

For creating the hardware, we were also unable to measure accurate power consumption of the resulting circuit. This is something that would be worth exploring, in particular for embedded devices. In addition to this, if the hardware could be generalised to not require a fixed input size, it would instantly become more useful for general-purpose computers.

Finally we suggest the repetition of this experiment, making some key changes to the methodology. We first suggest using larger networks, and the advanced pixel-correlation knowledge distillation for super-resolution networks (He et al., 2020). Secondly, we propose the repetition of this experiment with extreme quantised distillation, where the teacher network is reminiscent of GigaGAN or ESRT (Lu et al., 2022; Kang et al., 2023).

WORD COUNT: 10,667

Bibliography

Aarrestad, T. et al., 2021. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. learn. sci. tech.* [Online], 2(4), p.045015. 2101.05108, Available from: https://doi.org/10.1088/2632-2153/ac0ea1.

Abdelfattah, M., 2022a. Machine learning hardware and systems, lecture 11: Kernel computation [Online]. Available from: https://abdelfattah-class.github.io/ece5545/.

Abdelfattah, M., 2022b. Machine learning hardware and systems, lecture 2: Ml hardware [Online]. Available from: https://abdelfattah-class.github.io/ece5545/.

Abdelfattah, M., 2022c. Machine learning hardware and systems, lecture 7: Quantization [Online]. Available from: https://abdelfattah-class.github.io/ece5545/.

Agustsson, E. and Timofte, R., 2017. Ntire 2017 challenge on single image super-resolution: Dataset and study [Online]. *2017 ieee conference on computer vision and pattern recognition workshops (cvprw)*. pp.1122–1131. Available from: https://doi.org/10.1109/CVPRW.2017.150.

AMD, n.d.a. Amd virtex ultrascale+ fpga vcu118 evaluation kit [Online]. Available from: https://www.xilinx.com/products/boards-and-kits/vcu118.html [Accessed 19 April 2024].

AMD, n.d.b. Aup pynq-z2 [Online]. Available from: https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html [Accessed 19 April 2024].

AMD, n.d.c. Kria k26 system-on-module commercial [Online]. Available from: https://www.amd.com/en/products/system-on-modules/kria/k26/k26c-commercial.html [Accessed 19 April 2024].

AMD, n.d.d. Mercury+ xu7 zynq ultrascale+ mpsoc som/zu9eg [Online]. Available from: https://www.xilinx.com/products/boards-and-kits/1-1hp62i0.html [Accessed 19 April 2024].

AMD, n.d.e. Virtex ultrascale+ 56g pam4 vcu129 fpga evaluation kit [Online]. Available from: https://www.xilinx.com/products/boards-and-kits/vcu129.html [Accessed 19 April 2024].

Angarano, S., Salvetti, F., Martini, M. and Chiaberge, M., 2023. Generative adversarial super-resolution at the edge with knowledge distillation. *Engineering applications of artificial intelligence* [Online], 123, p.106407. Available from: https://doi.org/https://doi.org/10.1016/j.engappai.2023.106407.

Angelini, C., 2011. Intel core i7-3960x review: Sandy bridge-e and x79 express [Online]. Available from: https://www.tomshardware.com/reviews/core-i7-3960x-x79-sandy-bridge-e,3071.html [Accessed 12 April 2024].

Asif, U., Tang, J. and Harrer, S., 2020. Ensemble knowledge distillation for learning improved and efficient networks. 1909.08097.

au2, C.N.C.J., Kuusela, A., Li, S., Zhuang, H., Aarrestad, T., Loncar, V., Ngadiuba, J., Pierini, M., Pol, A.A. and Summers, S., 2021. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. 2006.10159.

auphelia, 2023. Does finn support prunning? [Online]. Available from: https://github.com/Xilinx/finn/issues/792#issuecomment-1507002305 [Accessed 21 April 2024].

avrumw, 2021. System logic cell, lut count on zynq fpga [Online]. Available from: https://support.xilinx.com/s/question/0D52E00007G0tfKSAR/system-logic-cell-lut-count-on-zynq-fpga [Accessed 19 April 2024].

Bevilacqua, M., Roumy, A., Guillemot, C.M. and Alberi-Morel, M.L., 2012. Low-complexity single-image super-resolution based on nonnegative neighbor embedding [Online]. *British machine vision conference*. Available from: https://doi.org/10.5244/C.26.135.

Blott, M., Preusser, T., Fraser, N., Gambardella, G., O'Brien, K. and Umuroglu, Y., 2018. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. 1809.04570.

Buciluundefined, C., Caruana, R. and Niculescu-Mizil, A., 2006. Model compression [Online]. *Proceedings of the 12th acm sigkdd international conference on knowledge discovery and data mining*. New York, NY, USA: Association for Computing Machinery, KDD '06, p.535–541. Available from: https://doi.org/10.1145/1150402.1150464.

Chai, J., Zeng, H., Li, A. and Ngai, E.W., 2021. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine learning with applications* [Online], 6, p.100134. Available from: https://doi.org/https://doi.org/10.1016/j.mlwa.2021.100134.

Clover, J., 2023. Apple silicon: The complete guide [Online]. Available from: https://www.macrumors.com/guide/apple-silicon/ [Accessed 30 December 2023].

Colbert, I., Pappalardo, A. and Petri-Koenig, J., 2023. A2q: Accumulator-aware quantization with guaranteed overflow avoidance. 2308.13504.

Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* [Online], 2(4), pp.303–314. Available from: https://doi.org/10.1007/BF02551274.

Daiphys, 2023. Finn | fpga : Xilinx : Tools - daiphys tech portal [Online]. Available from: https://www.daiphys.com/portal/fpga/xilinx/tools/finn.html [Accessed 15 March 2024].

Dally, B., 2018. Hardware for deep learning [Online]. Available from: https://www.youtube.com/watch?v=zDBF0xwQW-0.

Dally, W.J., Turakhia, Y. and Han, S., 2020. Domain-specific hardware accelerators. *Commun. acm* [Online], 63(7), p.48–57. Available from: https://doi.org/10.1145/3361682.

Davidson, J.W. and Jinturkar, S., 1994. Memory access coalescing: a technique for eliminating redundant memory accesses [Online]. *Proceedings of the acm sigplan 1994 conference on programming language design and implementation*. New York, NY, USA: Association for Computing Machinery, PLDI '94, p.186–195. Available from: https://doi.org/10.1145/178243.178259.

Deci, 2023. The correct way to measure inference time of deep neural networks [Online]. Available from: https://deci.ai/blog/measure-inference-time-deep-neural-networks/.

Dihuni, 2024. Nvidia a100 80gb gpu pci-e 4.0 [Online]. Available from: https://www.dihuni.com/product/nvidia-a100-80gb-900-21001-0020-000-gpu-pci-e-4-0/.

Dong, C., Loy, C.C., He, K. and Tang, X., 2015. Image super-resolution using deep convolutional networks. 1501.00092.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. and Houlsby, N., 2021. An image is worth 16x16 words: Transformers for image recognition at scale. 2010.11929.

Duarte, J. et al., 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Jinst* [Online], 13(07), p.P07027. 1804.06913, Available from: https://doi.org/10.1088/1748-0221/13/07/P07027.

Duchi, J., Hazan, E. and Singer, Y., 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. mach. learn. res.*, 12(null), p.2121–2159.

Dumoulin, V., Belghazi, I., Poole, B., Mastropietro, O., Lamb, A., Arjovsky, M. and Courville, A., 2017. Adversarially learned inference. 1606.00704.

Ghielmetti, N. et al., 2022. Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml. *Mach. learn. sci. tech.* [Online]. 2205.07690, Available from: https://doi.org/10.1088/2632-2153/ac9cb5.

Ghorpade, J., 2012. Gpgpu processing in cuda architecture. *Advanced computing: An international journal* [Online], 3(1), p.105–120. Available from: https://doi.org/10.5121/acij.2012.3109.

Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial networks. 1406.2661.

Gribbon, K., Johnston, C. and Bailey, D., 2003. A real-time fpga implementation of a barrel distortion correction algorithm with bilinear interpolation. *Proc. of image and vision computing new zealand conference*.

Gupta, S., Agrawal, A., Gopalakrishnan, K. and Narayanan, P., 2015. Deep learning with limited numerical precision [Online]. In: F. Bach and D. Blei, eds. *Proceedings of the 32nd international conference on machine learning*. Lille, France: PMLR, *Proceedings of Machine Learning Research*, vol. 37, pp.1737–1746. Available from: https://proceedings.mlr.press/v37/gupta15.html.

Hamanaka, F., Odan, T., Kise, K. and Chu, T.V., 2023. An exploration of state-of-the-art automation frameworks for fpga-based dnn acceleration. *Ieee access* [Online], 11, pp.5701–5713. Available from: https://doi.org/10.1109/ACCESS.2023.3236974.

Han, S., 2017. Efficient methods and hardware for deep learning [Online]. Available from: `https://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf` [Accessed 2 March 2024].

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

Han, S., Pool, J., Tran, J. and Dally, W.J., 2015. Learning both weights and connections for efficient neural networks. `1506.02626`.

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition [Online]. *2016 ieee conference on computer vision and pattern recognition (cvpr)*. pp.770–778. Available from: `https://doi.org/10.1109/CVPR.2016.90`.

He, Z., Dai, T., Lu, J., Jiang, Y. and Xia, S.T., 2020. Fakd: Feature-affinity based knowledge distillation for efficient image super-resolution [Online]. *2020 ieee international conference on image processing (icip)*. pp.518–522. Available from: `https://doi.org/10.1109/ICIP40778.2020.9190917`.

Hennessy, J., Patterson, D. and Asanović, K., 2012. *Computer architecture: A quantitative approach* [Online], Computer Architecture: A Quantitative Approach. Kaufmann. Available from: `https://books.google.co.uk/books?id=v3-1hVwHnHwC`.

Hinton, G. and Tieleman, T., 2012. RMSProp. Coursera course on neural networks. Lecture 6, Slide 29.

Hinton, G., Vinyals, O. and Dean, J., 2015. Distilling the knowledge in a neural network. `1503.02531`.

Ho, J., Jain, A. and Abbeel, P., 2020. Denoising diffusion probabilistic models. `2006.11239`.

Hornik, K., 1991. Approximation capabilities of multilayer feedforward networks. *Neural networks* [Online], 4(2), pp.251–257. Available from: `https://doi.org/https://doi.org/10.1016/0893-6080(91)90009-T`.

Horowitz, M., 2014. 1.1 computing's energy problem (and what we can do about it) [Online]. *2014 ieee international solid-state circuits conference digest of technical papers (isscc)*. pp.10–14. Available from: `https://doi.org/10.1109/ISSCC.2014.6757323`.

Horé, A. and Ziou, D., 2010. Image quality metrics: Psnr vs. ssim [Online]. *2010 20th international conference on pattern recognition*. pp.2366–2369. Available from: `https://doi.org/10.1109/ICPR.2010.579`.

Huang, J.B., Singh, A. and Ahuja, N., 2015. Single image super-resolution from transformed self-exemplars [Online]. *2015 ieee conference on computer vision and pattern recognition (cvpr)*. pp.5197–5206. Available from: `https://doi.org/10.1109/CVPR.2015.7299156`.

Huang, Q., Zhang, Y., Hu, H., Zhu, Y. and Zhao, Z., 2021. Binarizing super-resolution networks by pixel-correlation knowledge distillation [Online]. *2021 ieee international conference on image processing (icip)*. pp.1814–1818. Available from: `https://doi.org/10.1109/ICIP42928.2021.9506517`.

Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J. and Keutzer, K., 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. `1602.07360`.

Ignatov, A., Timofte, R., Denna, M. and Younes, A., 2021. Real-time quantized image super-resolution on mobile npus, mobile ai 2021 challenge: Report. *Proceedings of the ieee/cvf conference on computer vision and pattern recognition (cvpr) workshops*. pp.2525–2534.

Jia, Y., 2014. Learning semantic image representations at a large scale [Online]. Available from: `https://escholarship.org/uc/item/64c2v6sn` [Accessed 15 April 2024].

Kang, M., Zhu, J.Y., Zhang, R., Park, J., Shechtman, E., Paris, S. and Park, T., 2023. Scaling up gans for text-to-image synthesis. `2303.05511`.

Kastner, R., Matai, J. and Neuendorffer, S., 2018. Parallel Programming for FPGAs. *Arxiv e-prints*. `1805.03648`.

Kaz Sato, C.Y., 2017. An in-depth look at google's first tensor processing unit (tpu) [Online]. Available from: `https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu` [Accessed 2 December 2023].

Kim, M. and Smaragdis, P., 2016. Bitwise neural networks. `1601.06071`.

Kim, Y., Choi, J.S. and Kim, M., 2019. A real-time convolutional neural network for super-resolution on fpga with applications to 4k uhd 60 fps video services. *Ieee transactions on circuits and systems for video technology* [Online], 29(8), pp.2521–2534. Available from: `https://doi.org/10.1109/TCSVT.2018.2864321`.

Kingma, D.P. and Ba, J., 2017. Adam: A method for stochastic optimization. `1412.6980`.

Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2017. Imagenet classification with deep convolutional neural networks. *Commun. acm* [Online], 60(6), p.84–90. Available from: `https://doi.org/10.1145/3065386`.

Lavin, A. and Gray, S., 2015. Fast algorithms for convolutional neural networks. `1509.09308`.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W. and Jackel, L., 1989. Handwritten digit recognition with a back-propagation network [Online]. In: D. Touretzky, ed. *Advances in neural information processing systems*. Morgan-Kaufmann, vol. 2. Available from: `https://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf`.

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. *Proceedings of the ieee* [Online], 86(11), pp.2278–2324. Available from: `https://doi.org/10.1109/5.726791`.

Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z. and Shi, W., 2017. Photo-realistic single image super-resolution using a generative adversarial network. `1609.04802`.

Lee, S., Joo, S., Ahn, H.K. and Jung, S.O., 2020. Cnn acceleration with hardware-efficient dataflow for super-resolution. *Ieee access* [Online], 8, pp.187754–187765. Available from: `https://doi.org/10.1109/ACCESS.2020.3031055`.

Lu, Z., Li, J., Liu, H., Huang, C., Zhang, L. and Zeng, T., 2022. Transformer for single image super-resolution. `2108.11084`.

Ma, S., Liu, Z., Chen, S., Huang, L., Guo, Y., Wang, Z. and Zhang, M., 2019. Coordinated dma: Improving the dram access efficiency for matrix multiplication. *Ieee transactions on parallel and distributed systems* [Online], 30(10), pp.2148–2164. Available from: https://doi.org/10.1109/TPDS.2019.2906891.

Ma, Y., Xiong, H., Hu, Z. and Ma, L., 2018. Efficient super resolution using binarized neural network. 1812.06378.

Mao, X., Shen, C. and Yang, Y.B., 2016. Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections [Online]. In: D. Lee, M. Sugiyama, U. Luxburg, I. Guyon and R. Garnett, eds. *Advances in neural information processing systems*. Curran Associates, Inc., vol. 29. Available from: https://proceedings.neurips.cc/paper_files/paper/2016/file/0ed9422357395a0d4879191c66f4faa2-Paper.pdf.

Martin, D., Fowlkes, C., Tal, D. and Malik, J., 2001. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics [Online]. *Proceedings eighth ieee international conference on computer vision. iccv 2001*. vol. 2, pp.416–423 vol.2. Available from: https://doi.org/10.1109/ICCV.2001.937655.

McCabe, C., 2019. How to measure power consumption [Online]. Available from: https://discuss.pynq.io/t/how-to-measure-power-consumption/523/8 [Accessed 2 March 2024].

Mead, C. and Conway, L., 1980. Introduction to vlsi systems. *Reading, ma, addison-wesley publishing co., 1980. 426 p.*, -1.

Minhas, M.S., 2021. Computational graphs in pytorch and tensorflow [Online]. Available from: https://towardsdatascience.com/computational-graphs-in-pytorch-and-tensorflow-c25cc40bdcd1 [Accessed 3 April 2024].

Nagel, M., Amjad, R.A., Baalen, M. van, Louizos, C. and Blankevoort, T., 2020. Up or down? adaptive rounding for post-training quantization. 2004.10568.

Ngadiuba, J. et al., 2021. Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML. *Mach. learn. sci. tech.* [Online], 2, p.015001. 2003.06308, Available from: https://doi.org/10.1088/2632-2153/aba042.

Nikam, A., 2017. How is using im2col operation in convolutional nets more efficient? [Online]. Available from: https://stackoverflow.com/a/47422548 [Accessed 15 April 2024].

NVIDIA, n.d. Jetson nano [Online]. Available from: https://developer.nvidia.com/embedded/jetson-nano [Accessed 19 April 2024].

NVIDIA, 2024. Nvidia a100 [Online]. Available from: https://www.nvidia.com/en-gb/data-center/a100/.

Odena, A., Dumoulin, V. and Olah, C., 2016. Deconvolution and checkerboard artifacts. *Distill* [Online]. Available from: http://distill.pub/2016/deconv-checkerboard/.

O'Sullivan, C., 2023. U-net explained: Understanding its image segmentation architecture [Online]. Available from: https://towardsdatascience.com/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a.

Pang, Y., Lin, J., Qin, T. and Chen, Z., 2021. Image-to-image translation: Methods and applications. `2101.08629`.

Park, C., 2002. Neither fish nor fowl? the perceived benefits and problems of using graduate teaching assistants (gtas) to teach undergraduate students. *Higher education review*, 35, p.50.

Pettersson, L., 2020. *Convolutional neural networks on fpga and gpu on the edge: A comparison* [Online]. Ph.D. thesis. Uppsala University. Available from: `https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-414608`.

Polyak, B., 1964. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics* [Online], 4(5), pp.1–17. Available from: `https://doi.org/https://doi.org/10.1016/0041-5553(64)90137-5`.

Ras, L., 2018. Deep neural network skip connection implemented as summation vs concatenation? [Online]. Available from: `https://stackoverflow.com/a/49179305` [Accessed 13 April 2024].

Raschka, S., 2023. Finetuning llms on a single gpu using gradient accumulation [Online]. Available from: `https://lightning.ai/blog/gradient-accumulation/` [Accessed 15 March 2024].

Roberts, J., Crall, J., Ang, K.M. and Brandt, Y., 2024. *alan-turing-institute/distinctipy: v1.3.4* (v.v1.3.4). Zenodo. Available from: `https://doi.org/10.5281/zenodo.10480933`.

Ronneberger, O., Fischer, P. and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. `1505.04597`.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J., 1986. Learning Internal Representations by Error Propagation [Online]. *Parallel Distributed Processing, Volume 1: Explorations in the Microstructure of Cognition: Foundations*. The MIT Press. `https://direct.mit.edu/book/chapter-pdf/2163044/9780262291408_cah.pdf`, Available from: `https://doi.org/10.7551/mitpress/5236.003.0012`.

Saharia, C., Ho, J., Chan, W., Salimans, T., Fleet, D.J. and Norouzi, M., 2021. Image super-resolution via iterative refinement. `2104.07636`.

Schmittler, J., Woop, S., Wagner, D., Paul, W.J. and Slusallek, P., 2004. Realtime ray tracing of dynamic scenes on an fpga chip [Online]. *Proceedings of the acm siggraph/eurographics conference on graphics hardware*. New York, NY, USA: Association for Computing Machinery, HWWS '04, p.95–106. Available from: `https://doi.org/10.1145/1058129.1058143`.

Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A.P., Bishop, R., Rueckert, D. and Wang, Z., 2016. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. `1609.05158`.

Shin, S., Boo, Y. and Sung, W., 2020. Knowledge distillation for optimization of quantized deep neural networks [Online]. *2020 ieee workshop on signal processing systems (sips)*. pp.1–6. Available from: `https://doi.org/10.1109/SiPS50750.2020.9195219`.

Simonyan, K. and Zisserman, A., 2015. Very deep convolutional networks for large-scale image recognition. `1409.1556`.

Sohl-Dickstein, J., Weiss, E.A., Maheswaranathan, N. and Ganguli, S., 2015. Deep unsupervised learning using nonequilibrium thermodynamics. 1503.03585.

Strassen, V., 1969. Gaussian elimination is not optimal. *Numerische mathematik* [Online], 13, pp.354–356. Available from: http://eudml.org/doc/131927.

Su, Y., Seng, K.P., Smith, J. and Ang, L.M., 2024. Efficient fpga binary neural network architecture for image super-resolution. *Electronics* [Online], 13(2). Available from: https://doi.org/10.3390/electronics13020266.

Sun, K., Koch, M., Wang, Z., Jovanovic, S., Rabah, H. and Simon, S., 2022. An fpga-based residual recurrent neural network for real-time video super-resolution. *Ieee transactions on circuits and systems for video technology* [Online], 32(4), pp.1739–1750. Available from: https://doi.org/10.1109/TCSVT.2021.3080241.

TechPowerUp, 2024. Nvidia geforce rtx 3080 mobile [Online]. Available from: https://www.techpowerup.com/gpu-specs/geforce-rtx-3080-mobile.c3684 [Accessed 21 April 2024].

Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M. and Vissers, K., 2017. Finn: A framework for fast, scalable binarized neural network inference [Online]. *Proceedings of the 2017 acm/sigda international symposium on field-programmable gate arrays*. ACM, FPGA '17. Available from: https://doi.org/10.1145/3020078.3021744.

Umuroglu, Y. and Jahre, M., 2018. Streamlined deployment for quantized neural networks. 1709.04060.

UserBenchmark, 2024. Userbenchmark: Nvidia rtx 3060-ti vs 3080 (laptop) [Online]. Available from: https://gpu.userbenchmark.com/Compare/Nvidia-RTX-3080-Laptop-vs-Nvidia-RTX-3060-Ti/m1443565vs4090 [Accessed 11 April 2024].

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2023. Attention is all you need. 1706.03762.

Wang, Y.E., Wei, G.Y. and Brooks, D., 2019. Benchmarking tpu, gpu, and cpu platforms for deep learning. 1907.10701.

Wang, Z., Chen, J. and Hoi, S.C.H., 2020. Deep learning for image super-resolution: A survey. 1902.06068.

Winograd, S., 1978. On computing the discrete fourier transform. *Mathematics of computation* [Online], 32(141), pp.175–199. Available from: http://www.jstor.org/stable/2006266 [Accessed 2024-04-24].

Zeyde, R., Elad, M. and Protter, M., 2012. On single image scale-up using sparse-representations. In: J.D. Boissonnat, P. Chenin, A. Cohen, C. Gout, T. Lyche, M.L. Mazure and L. Schumaker, eds. *Curves and surfaces*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.711–730.

Zhang, R., Isola, P., Efros, A.A., Shechtman, E. and Wang, O., 2018. The unreasonable effectiveness of deep features as a perceptual metric. *Cvpr*.

Zhao, H., Gallo, O., Frosio, I. and Kautz, J., 2018. Loss functions for image restoration with neural networks. 1511.08861.

Zhou, D.X., 2020. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis* [Online], 48(2), pp.787–794. Available from: `https://doi.org/https://doi.org/10.1016/j.acha.2019.06.004`.

Appendix A

Technical Details

A.1 Image Similarity Metrics

The Peak Signal-to-Noise ratio measures the maximum possible power between two signals. For our case, how similar two images are. It is logarithmic, meaning the larger the value, the more noticeable the change is, and is measured in decibels. For example, an improvement from 31dB to 32dB is a lot better than the improvement from 20dB to 21dB. The equation below uses 255 to represent the maximum value of any integer in the image representation, but could also be 1.0 for floating point images. Notice that the PSNR tends towards infinity as the mean squared error of the two images tends toward zero (Horé and Ziou, 2010).

$$\text{PSNR}(x, y) = 10 \cdot \log_{10}\left(\frac{255^2}{\text{MSE}(x, y)}\right) \tag{A.1}$$

The Structural Similarity is a lot more complex to interpret, however it is just the combination of other metrics. It is the product of three functions. luminence(x, y) is the function which compares the mean luminence of the two images μ_x, μ_y. contrast(x, y) is the function comparing the closeness of the constrast between the two images, measured as the standard deviation σ_x, σ_y. Finally, the function structure(x, y) is the function that measures how the images relate to one another, with σ_{xy} being a measure of how similar the change in pixels is across images. The SSIM is a value between -1 and 1, where 1 represents perfect similarity, 0 represents none, and -1 represents an inverse relationship (Horé and Ziou, 2010).

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{A.2}$$

A.2 Gradient Accumulation Algorithm

Algorithm 1 Gradient Accumulation for Simulating Larger Batches (One Epoch)

```
1: **Input:** Training set, model parameters, optimizer, loss function, number of smaller batches
   to accumulate $batches\_to\_accum$
2:
3: **for** each mini-batch ($iteration, targets$) in training set **do**
4:     $outputs \leftarrow inference(mini - batch)$
5:     $loss\_val \leftarrow loss(outputs, targets)\ /\ batches\_to\_accum$
6:
7:     Backward propagate $loss\_val$ to compute gradients
8:
9:     **if** ($iteration + 1$) is a multiple of $batches\_to\_accum$ **then**
10:        Update model parameters using optimizer
11:
12:    **end if**
13: **end for**
```

A.3 Hardware

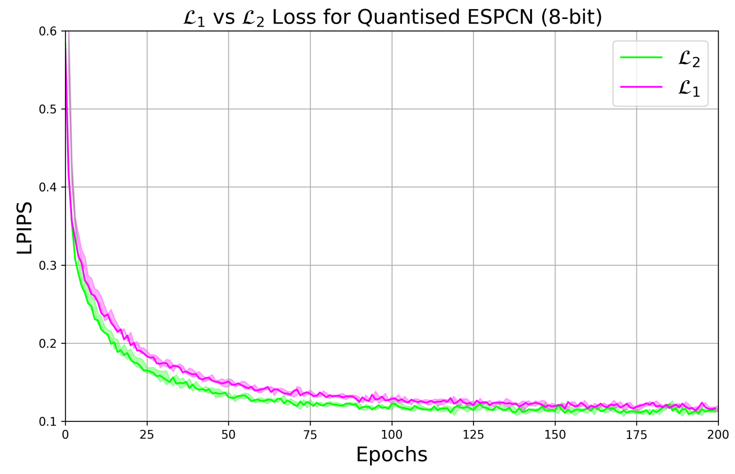
For training, and measuring GPU throughput, the NVIDIA RTX 3080 (Laptop) GPU was used as below.

NVIDIA CUDA Cores	6144
Memory Configuration	8GiB
Memory Interface Width	256-bit

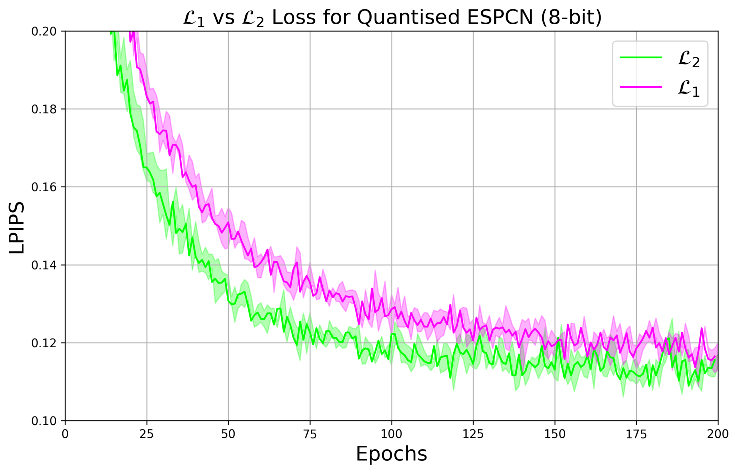
Appendix B

Detailed Results

B.1 Comparing Loss Functions



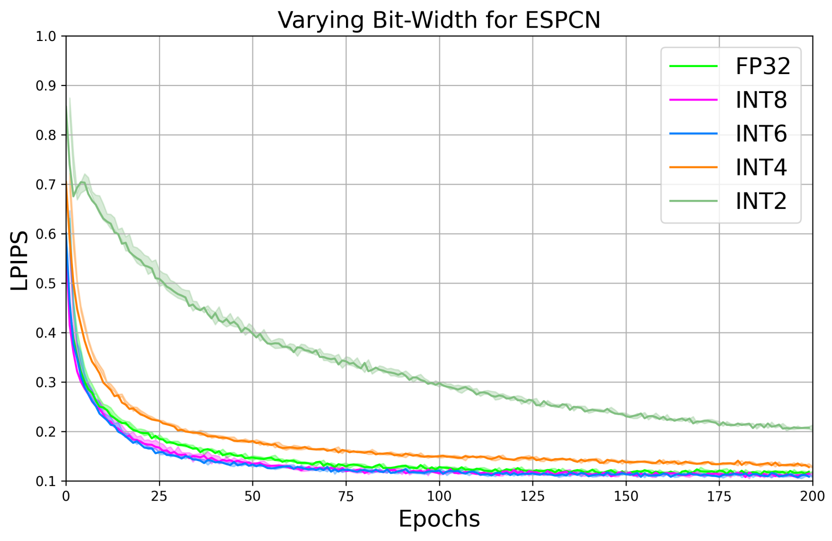
(a) Overall



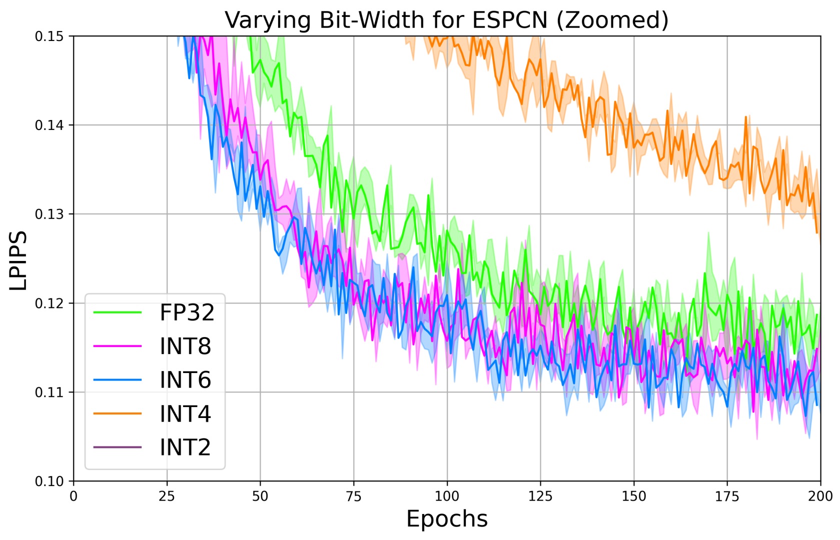
(b) Zoomed

Figure B.1: $\mathcal{L}_1$ vs $\mathcal{L}_2$ loss functions for QESPCN

B.2 Training Baseline Networks

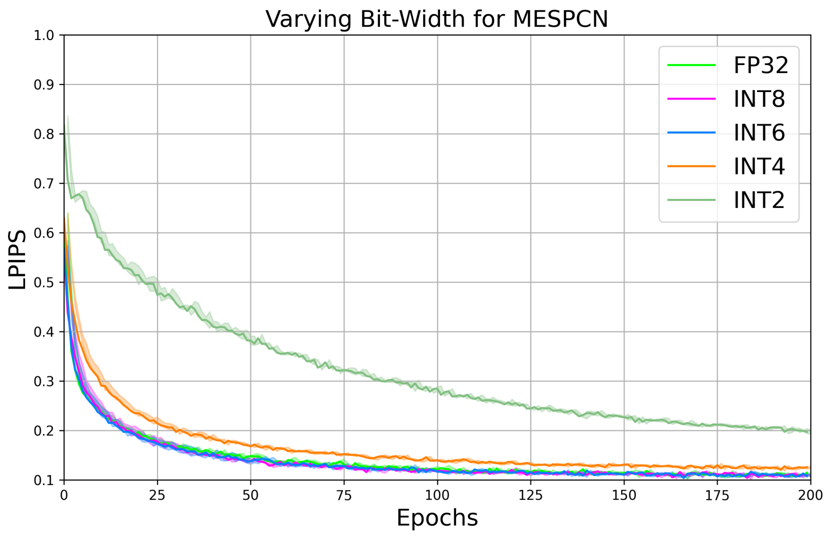


(a) ESPCN

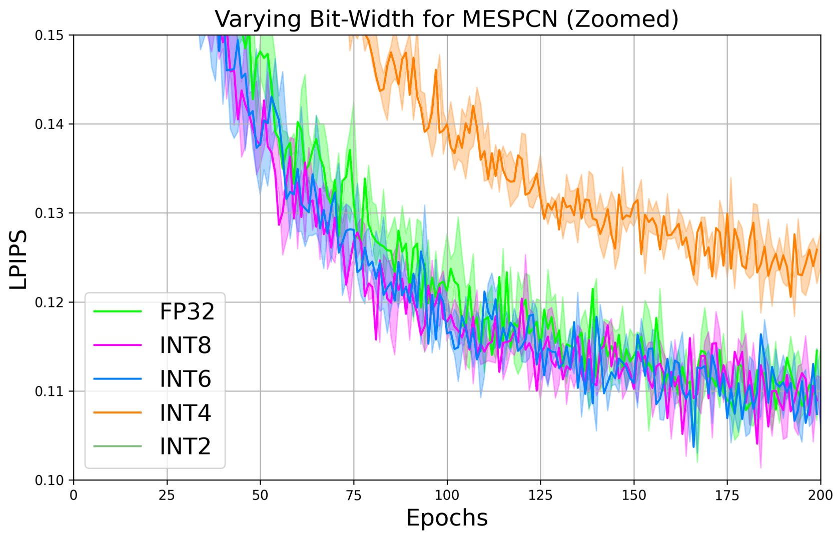


(b) ESPCN (Zoomed)

Figure B.2: Image similarity during training for ESPCN

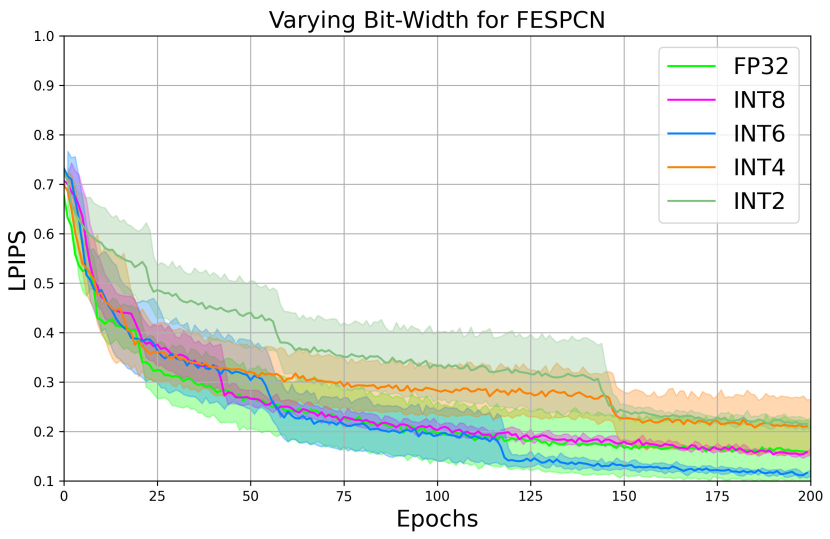


(a) MESPCN

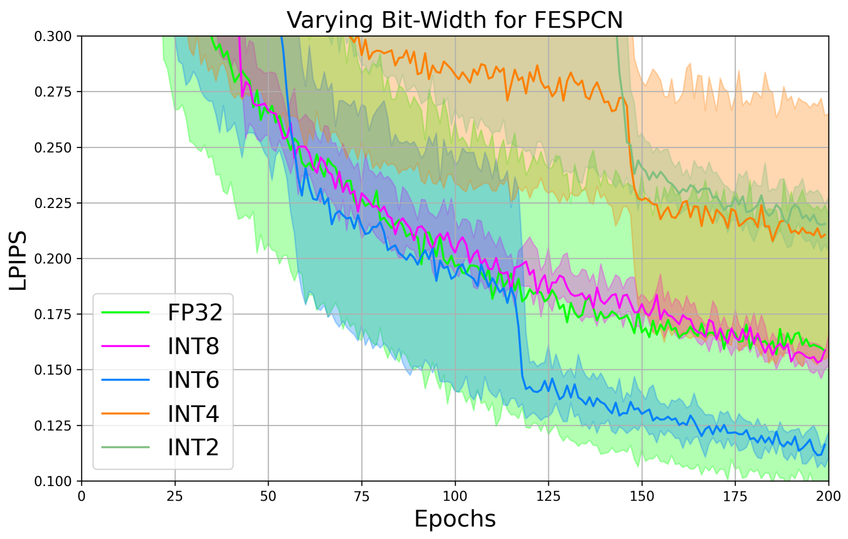


(b) MESPCN (Zoomed)

Figure B.3: Image similarity during training for MESPCN



(a) FESPCN



(b) FESPCN (Zoomed)

Figure B.4: Image similarity during training for FESPCN

B.3 Baseline Networks Test Set Performance

Best results per row are in **bold**, and worst results are <u>underlined</u>.

	ESPCN	MESPCN	FESPCN
BSD100	0.1587	**0.1537**	<u>0.1782</u>
Urban100	0.1469	**0.1411**	<u>0.1524</u>
Set5	0.0680	**0.0616**	<u>0.1302</u>
Set14	0.1098	**0.1040**	<u>0.1627</u>

Table B.1: Test set evaluation (average LPIPS scores) on FP32 networks

	ESPCN	MESPCN	FESPCN
BSD100	0.1529	**0.1511**	<u>0.1944</u>
Urban100	0.1413	**0.1379**	<u>0.1542</u>
Set5	0.0674	**0.0627**	<u>0.1296</u>
Set14	0.1058	**0.1024**	<u>0.1751</u>

Table B.2: Test set evaluation (average LPIPS scores) on INT8 networks

	ESPCN	MESPCN	FESPCN
BSD100	0.1514	<u>0.1519</u>	**0.1502**
Urban100	<u>0.1414</u>	0.1400	**0.1294**
Set5	0.0636	**0.0607**	<u>0.0726</u>
Set14	0.1026	**0.1021**	<u>0.1115</u>

Table B.3: Test set evaluation (average LPIPS scores) on INT6 networks

	ESPCN	MESPCN	FESPCN
BSD100	0.1655	**0.1606**	<u>0.2433</u>
Urban100	0.1522	**0.1495**	<u>0.1882</u>
Set5	0.0710	**0.0689**	<u>0.1866</u>
Set14	0.1158	**0.1125**	<u>0.2273</u>

Table B.4: Test set evaluation (average LPIPS scores) on INT4 networks

	ESPCN	MESPCN	FESPCN
BSD100	0.2200	**0.2148**	<u>0.2539</u>
Urban100	0.1948	**0.1919**	<u>0.1992</u>
Set5	**0.1360**	0.1374	<u>0.1685</u>
Set14	**0.1803**	0.1812	<u>0.2215</u>

Table B.5: Test set evaluation (average LPIPS scores) on INT2 networks

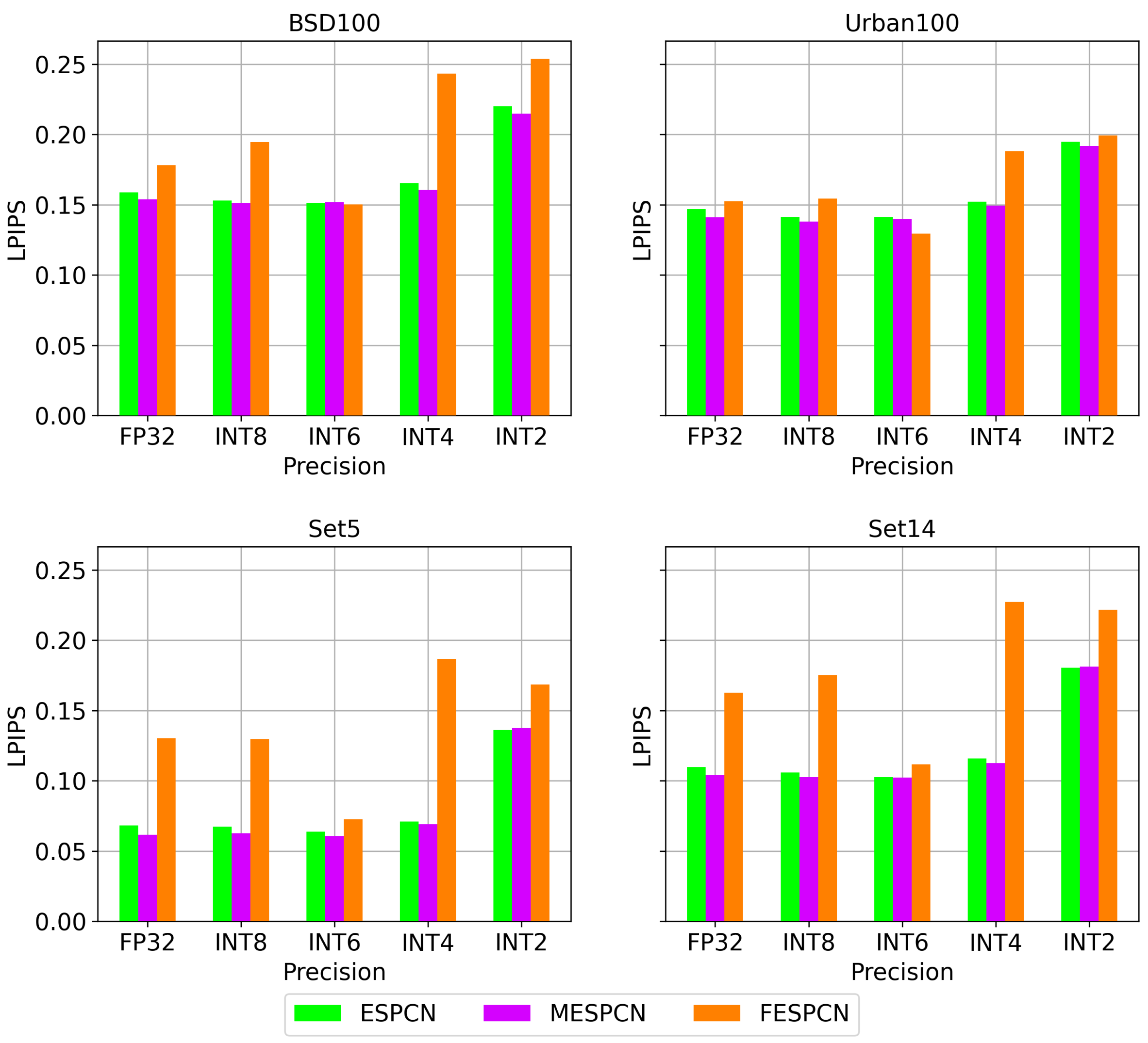


Figure B.5: Average LPIPS scores across test sets (lower is better)

B.4 Training Networks with Quantised Distillation

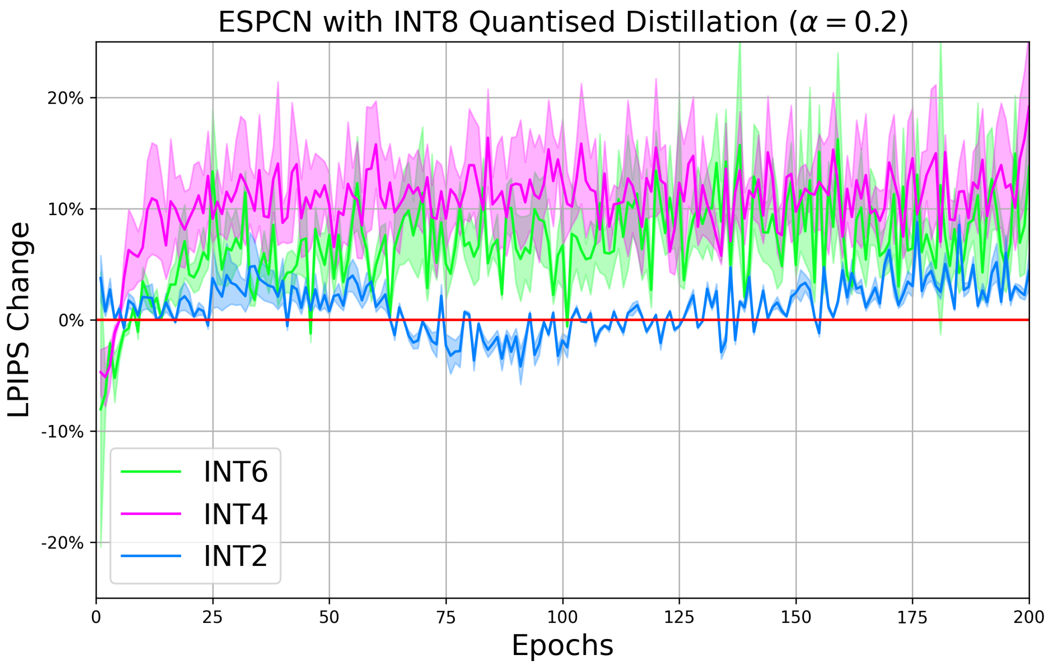


Figure B.6: Same architecture quantised distillation ($\alpha = 0.2$)

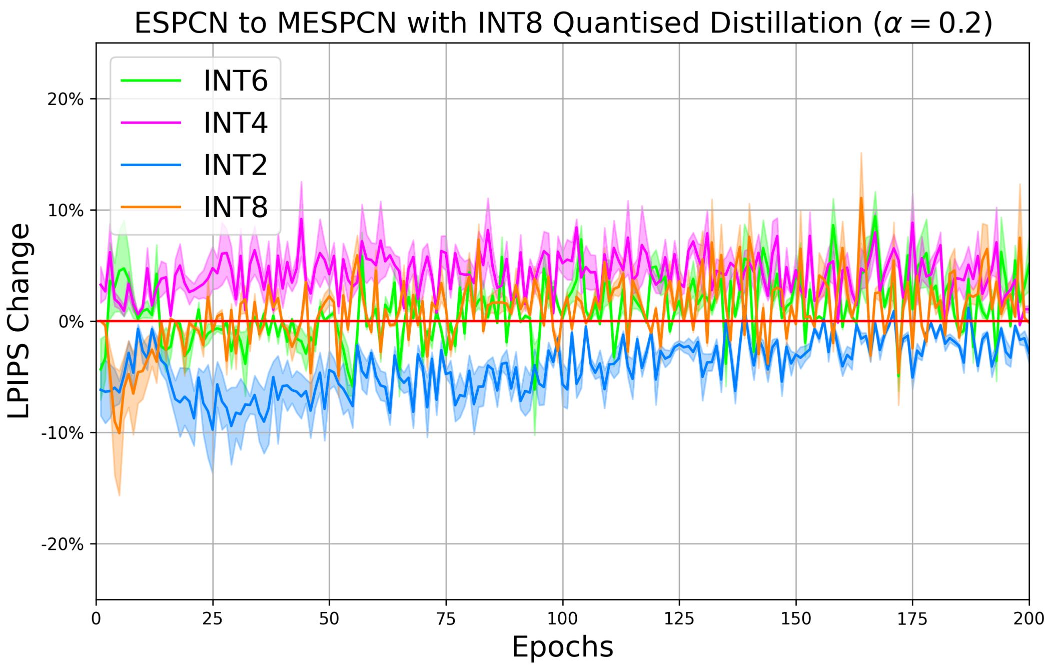


Figure B.7: Different architecture quantised distillation ($\alpha = 0.2$)

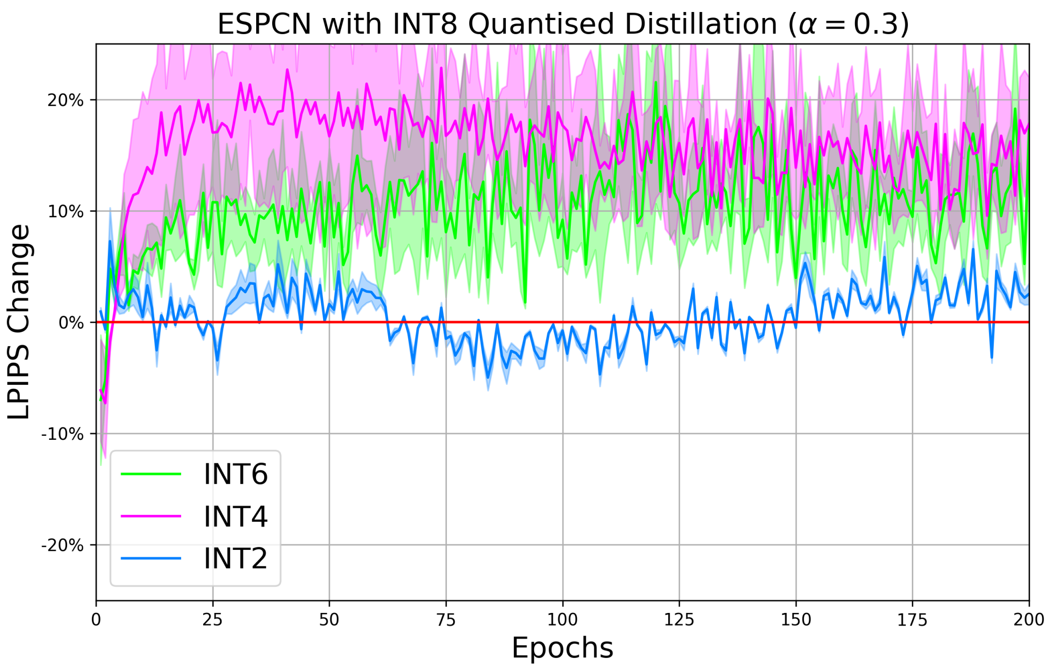


Figure B.8: Same architecture quantised distillation ($\alpha = 0.3$)

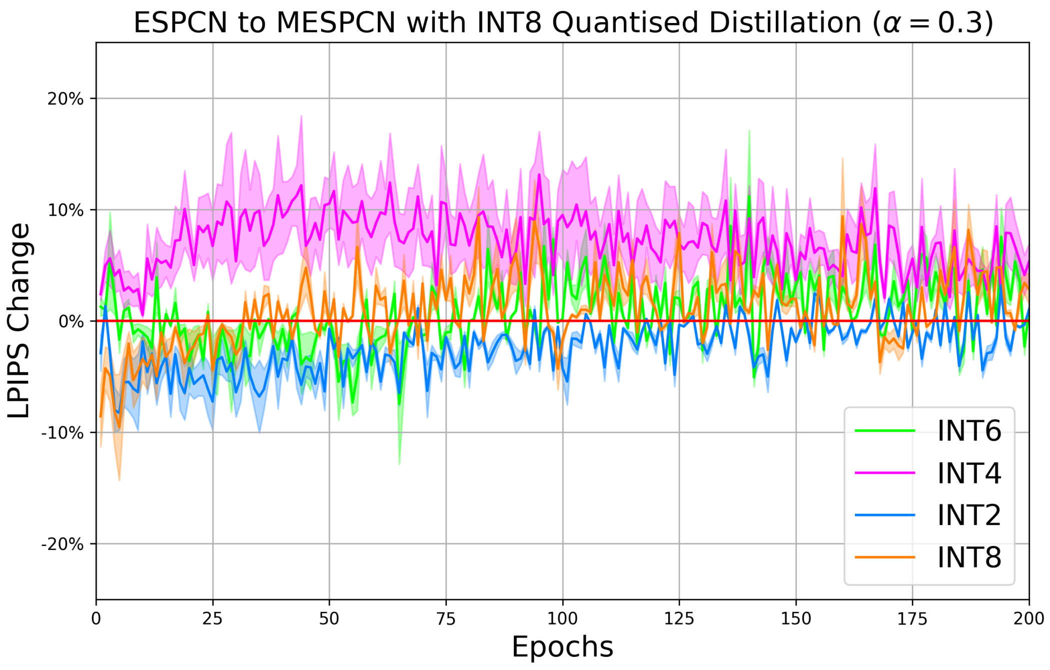


Figure B.9: Different architecture quantised distillation ($\alpha = 0.3$)

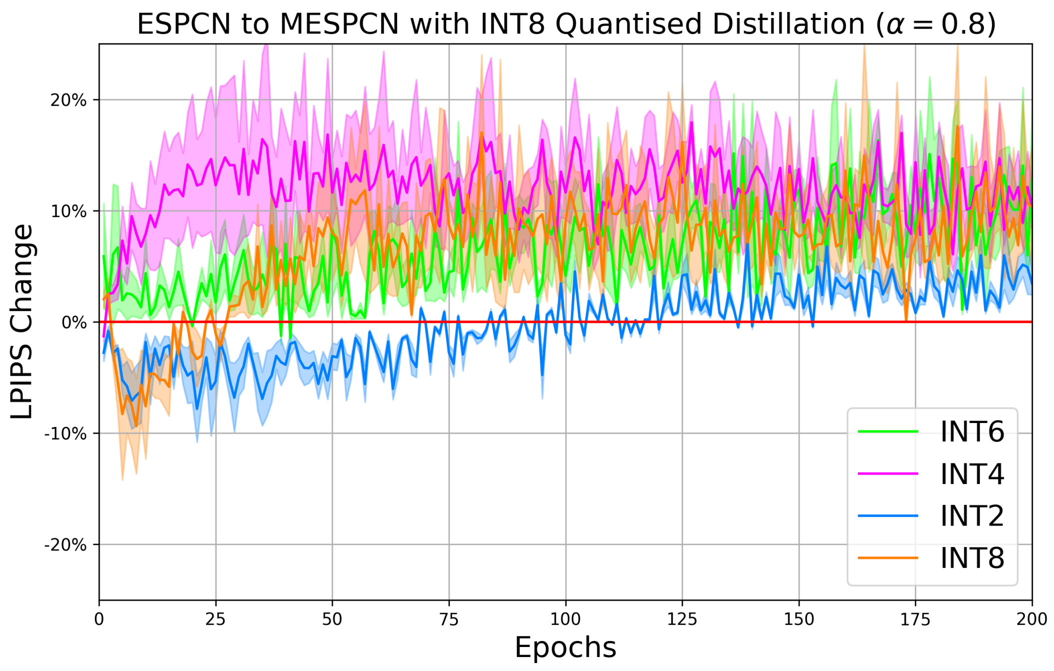


Figure B.10: Different architecture quantised distillation ($\alpha = 0.8$)

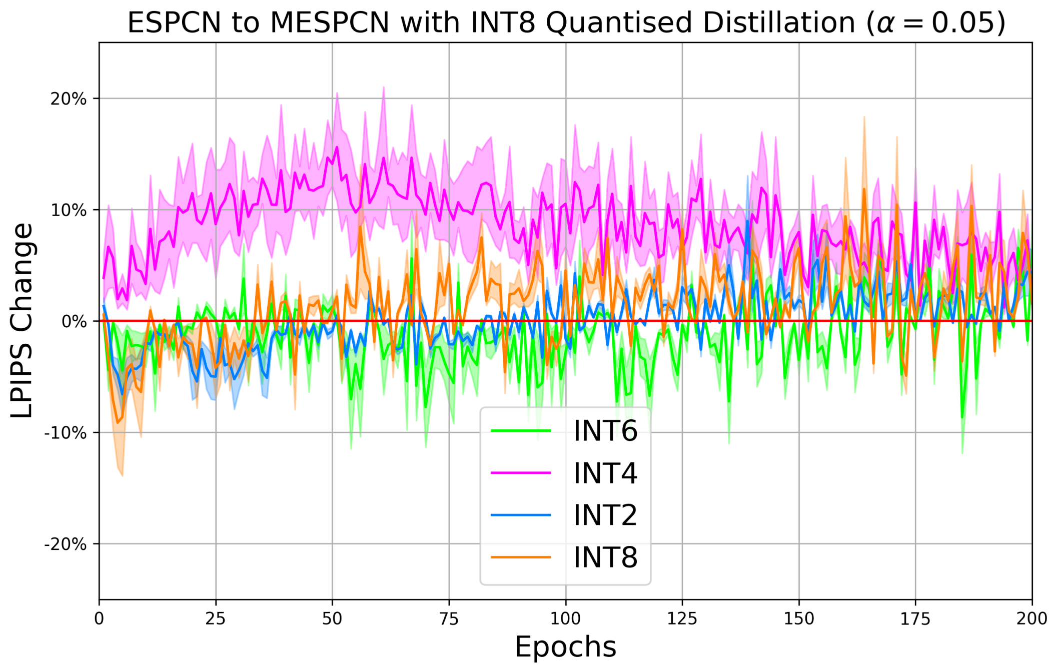


Figure B.11: Different architecture quantised distillation ($\alpha = 0.05$)

B.5 Quantised Distillation Test Set Performance

B.5.1 INT8 Networks

INT8-ESPCN was not trained with knowledge distillation, as it would be its own teacher. Therefore, it is not included in the graph below. Raw data for each of these graphs can be found in Appendix D.2.

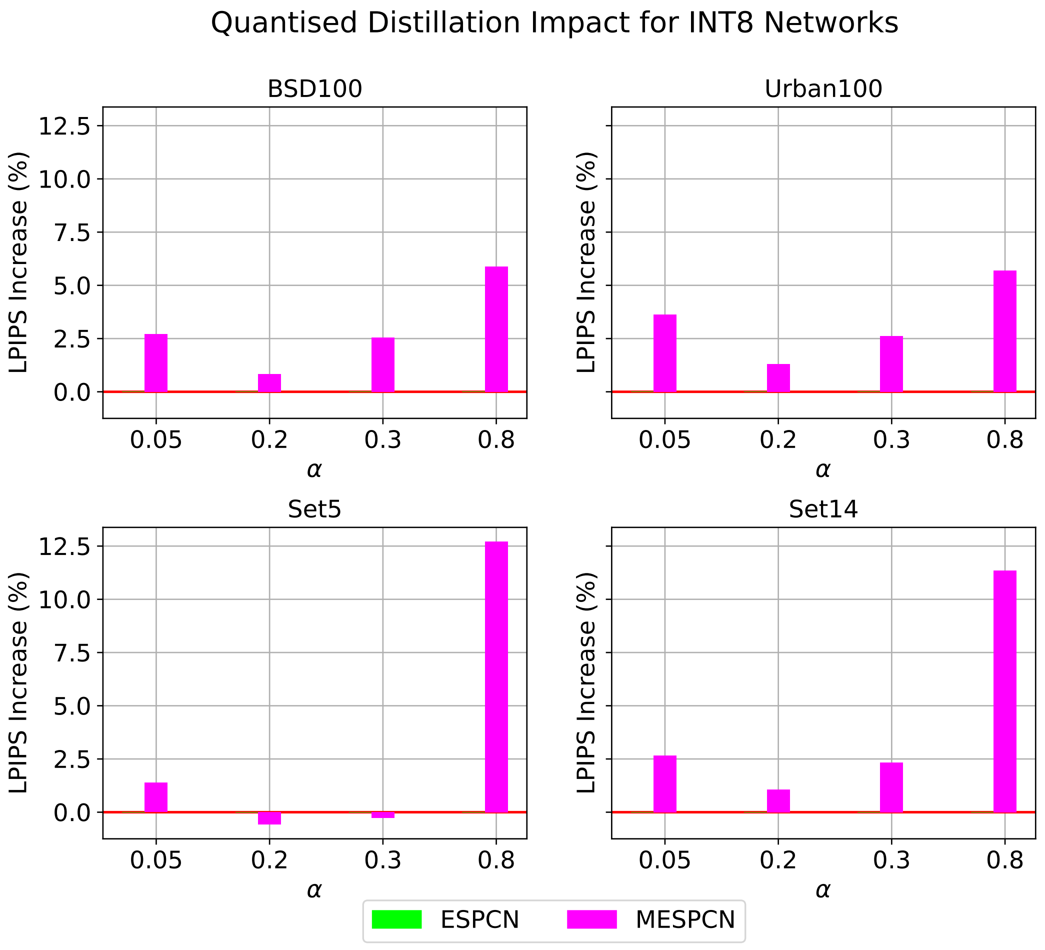


Figure B.12: Increase in LPIPS scores (INT8) (lower is better)

B.5.2 INT6 Networks

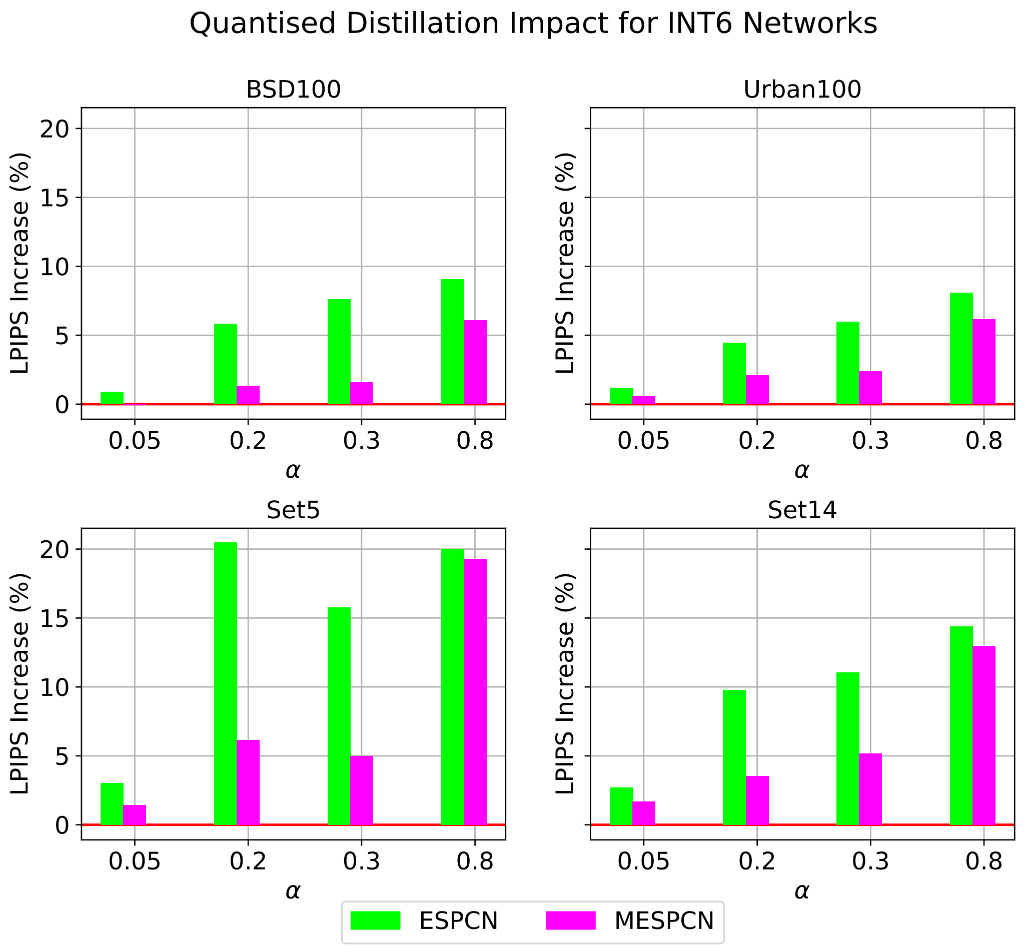


Figure B.13: Increase in LPIPS scores (INT6) (lower is better)

B.5.3 INT4 Networks

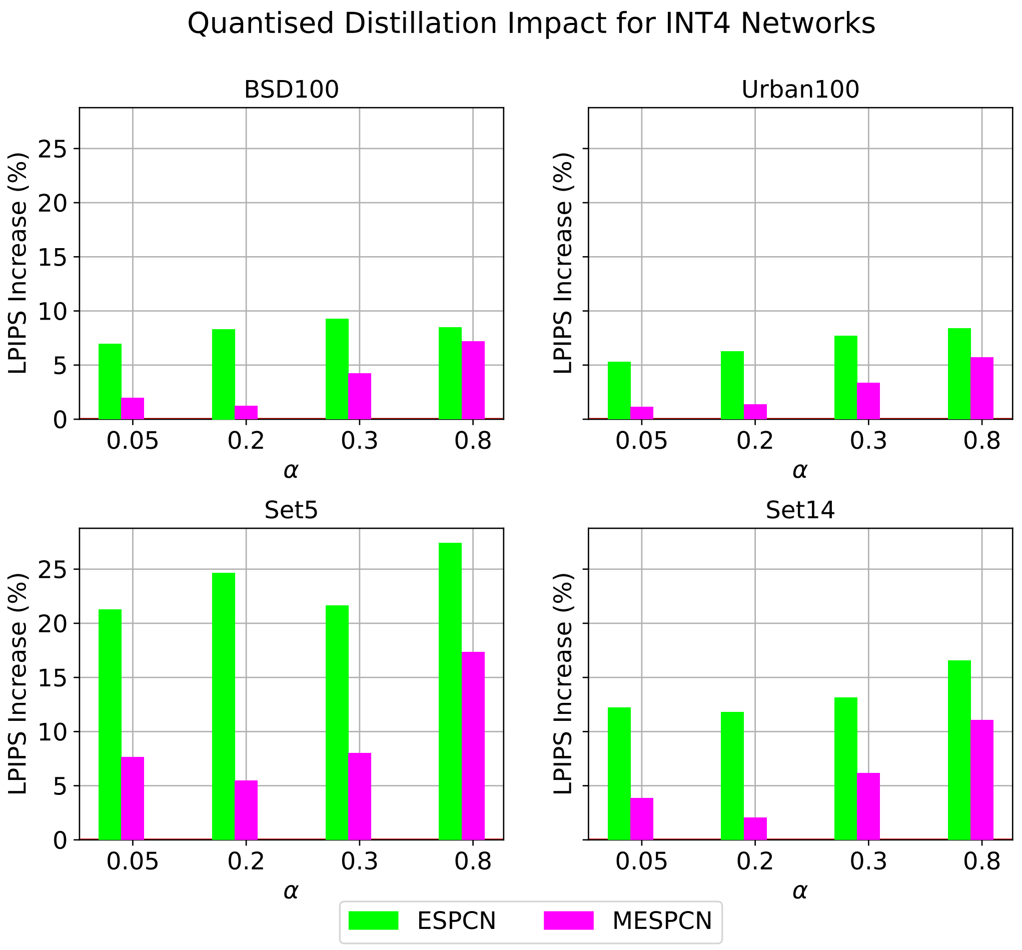


Figure B.14: Increase in LPIPS scores (INT4) (lower is better)

B.5.4 INT2 Networks

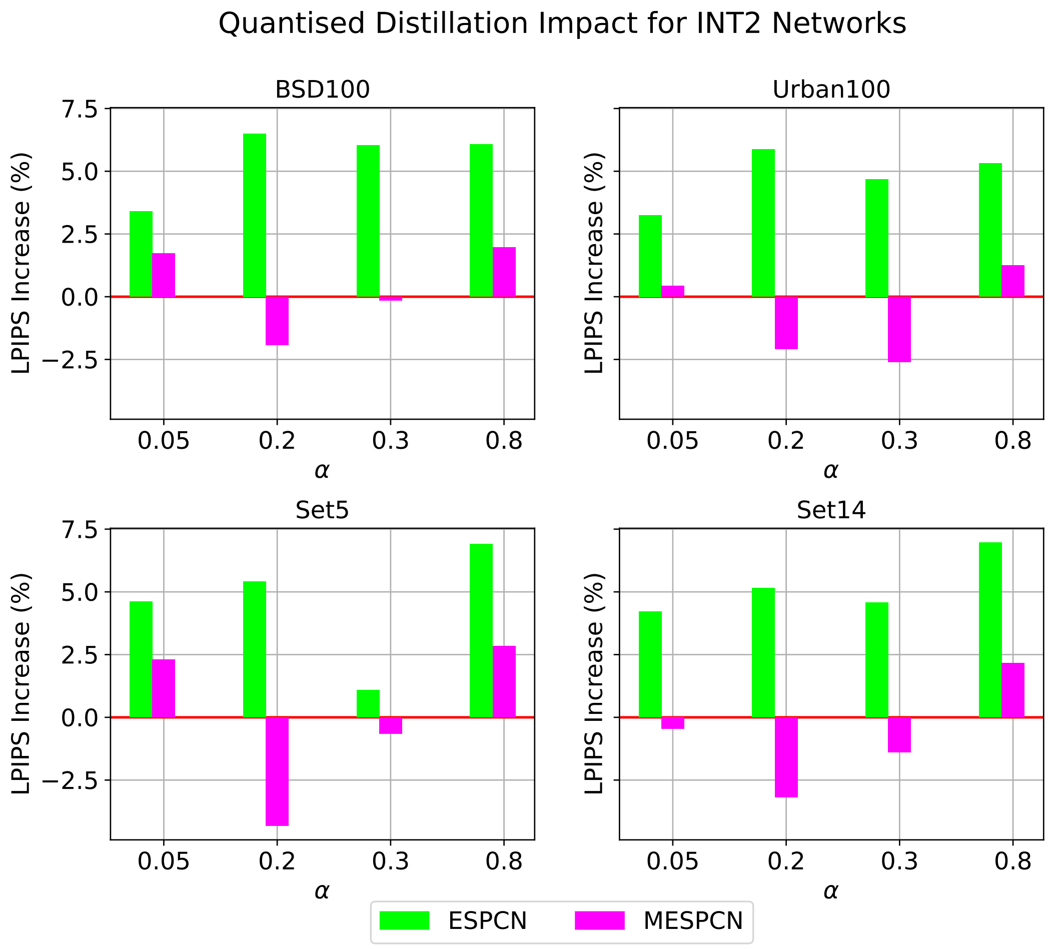


Figure B.15: Increase in LPIPS scores (INT2) (lower is better)

B.6 GPU Throughput per Network

The throughput of each of these networks was measured on an NVIDIA RTX 3080 (Laptop) graphics card, which is roughly equivalent to a NVIDIA RTX-3060-Ti within 3% effectives speed (UserBenchmark, 2024). The best result per row is in **bold**, and the worst is underlined.

	ESPCN	MESPCN	FESPCN
128 × 128px	2108	**3141**	1936
256 × 256px	571.8	**941.8**	547.3
512 × 512px	146.7	**239.7**	140.3

Table B.6: Average throughput (frames per second) of FP32 networks

Full data shown in Appendix D.1

Appendix C

Produced Images

C.1 ESPCN Produced Images (Baseline)

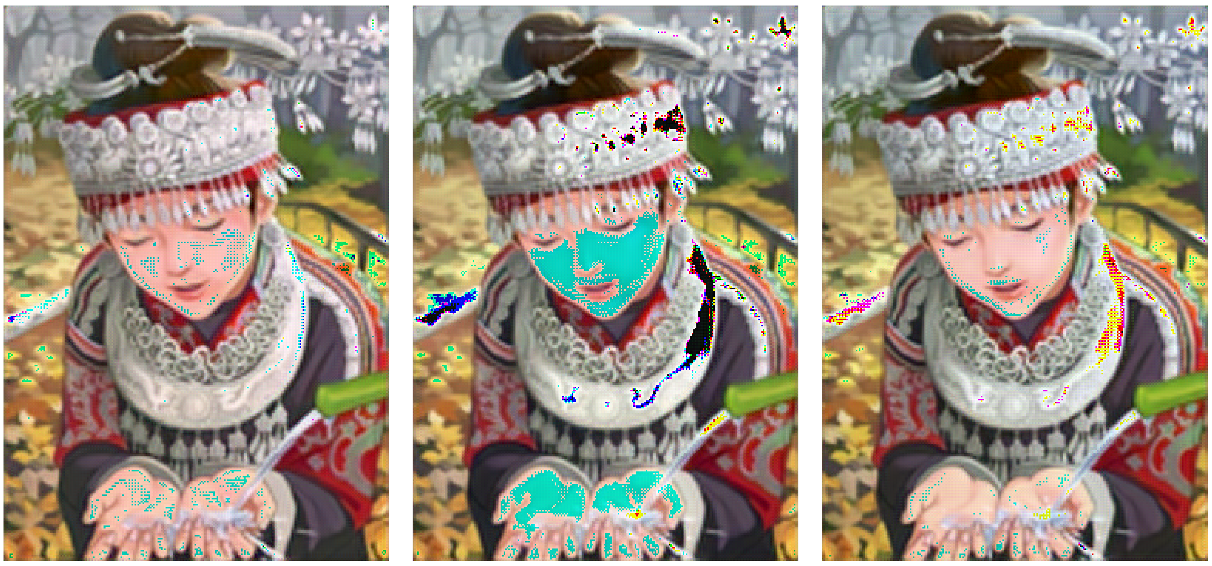


Figure C.1: FP32 ESPCN final images without knowledge distillation

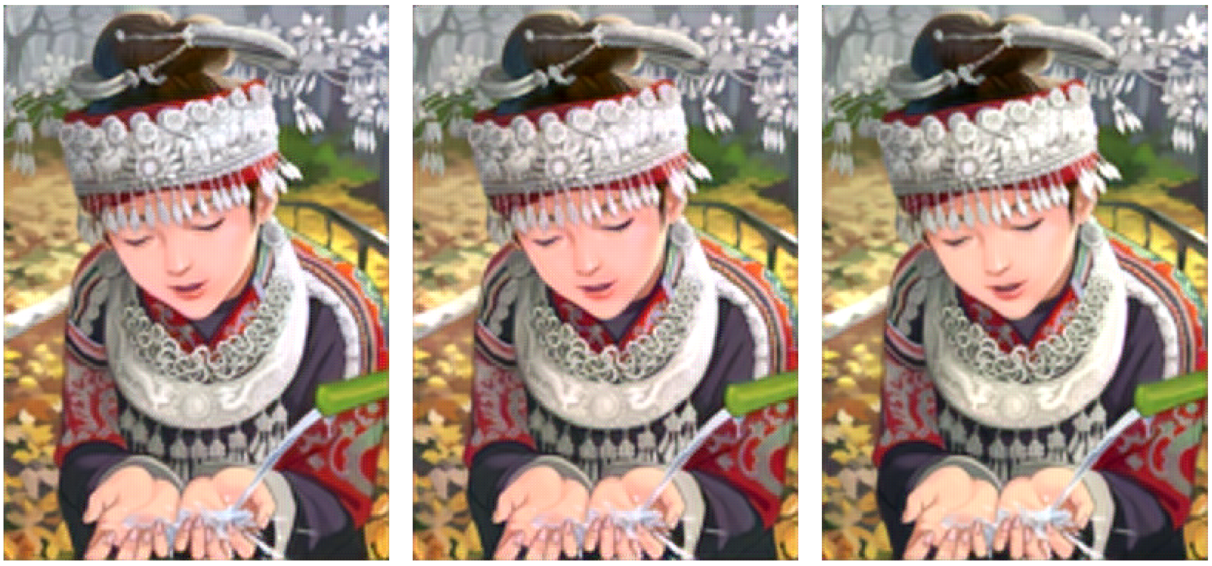


Figure C.2: INT8 ESPCN final images without knowledge distillation

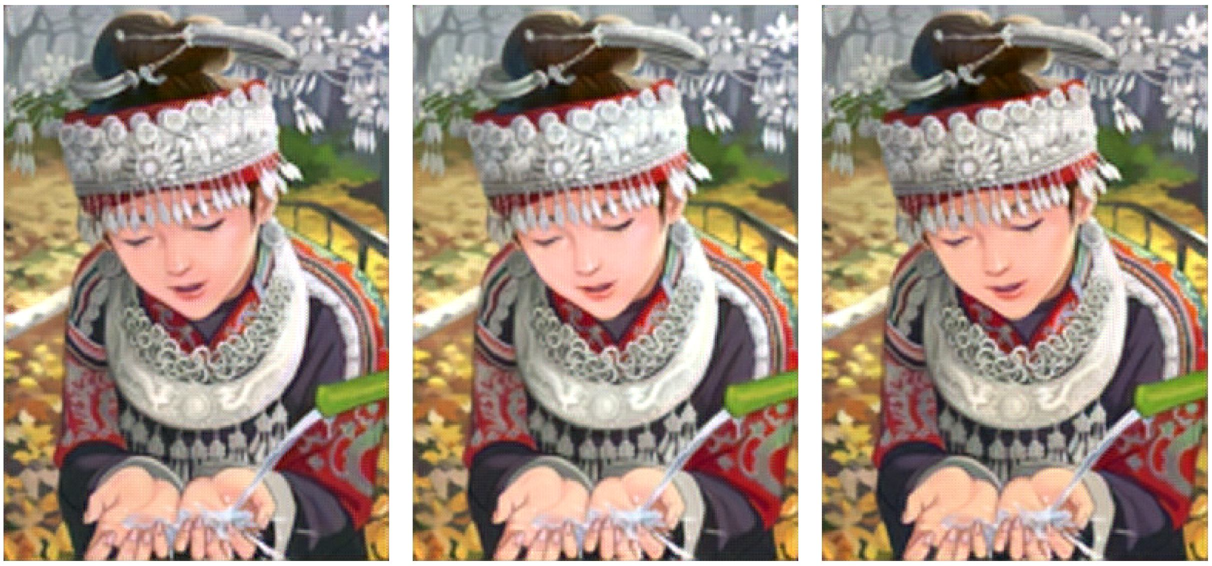


Figure C.3: INT6 ESPCN final images without knowledge distillation

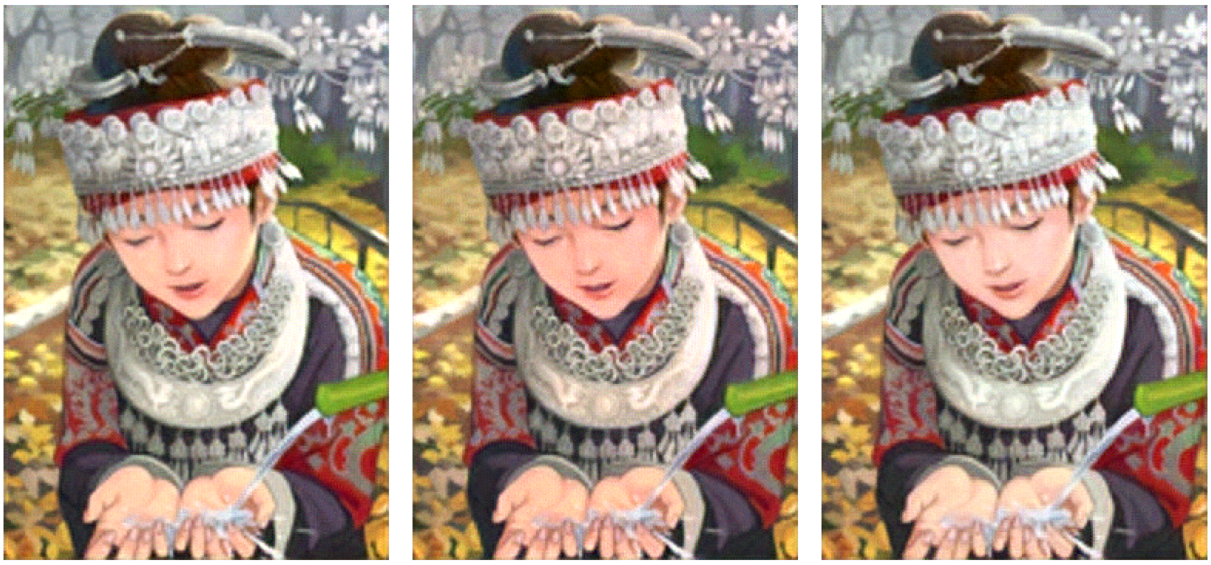


Figure C.4: INT4 ESPCN final images without knowledge distillation

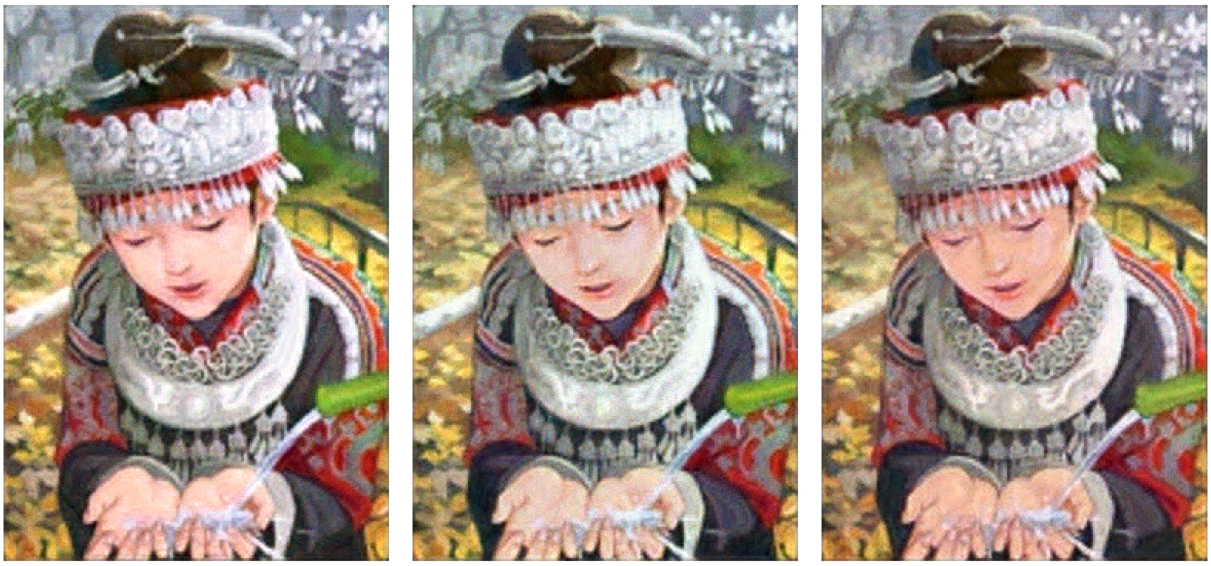


Figure C.5: INT2 ESPCN final images without knowledge distillation

C.2 MESPCN Produced Images ($\alpha = 0.2$)

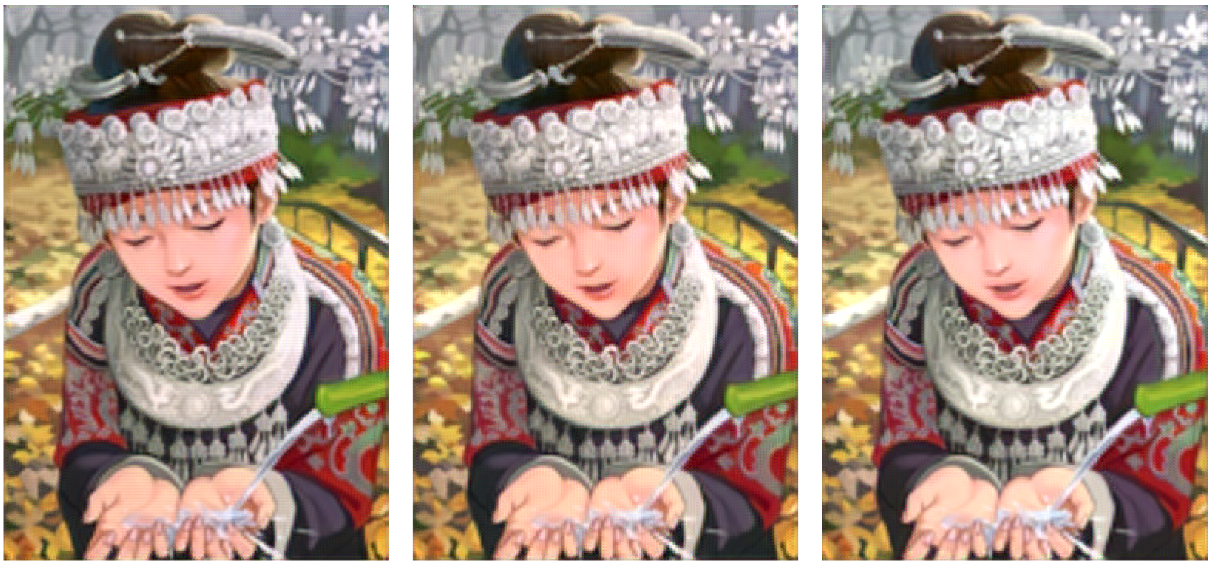


Figure C.6: INT8 MESPCN final images with knowledge distillation

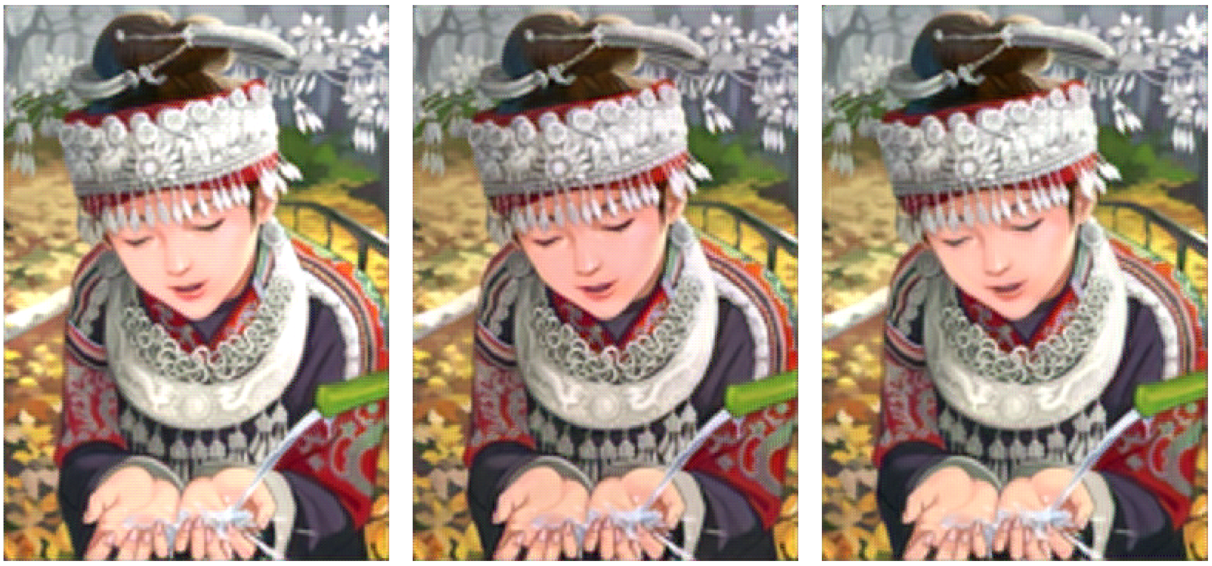


Figure C.7: INT6 MESPCN final images with knowledge distillation

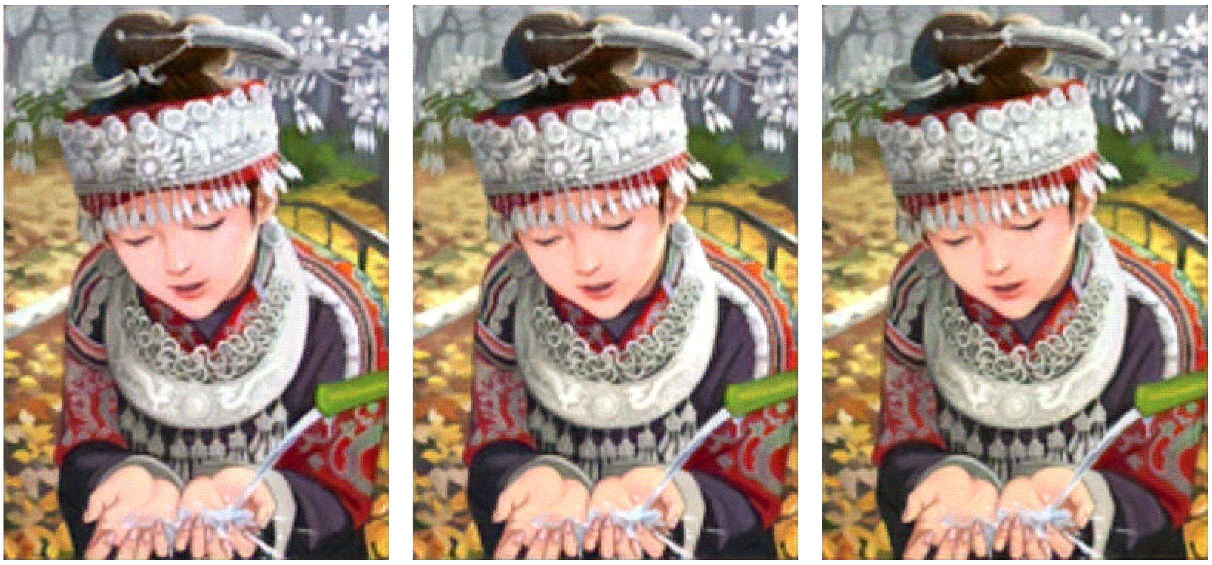


Figure C.8: INT4 MESPCN final images with knowledge distillation

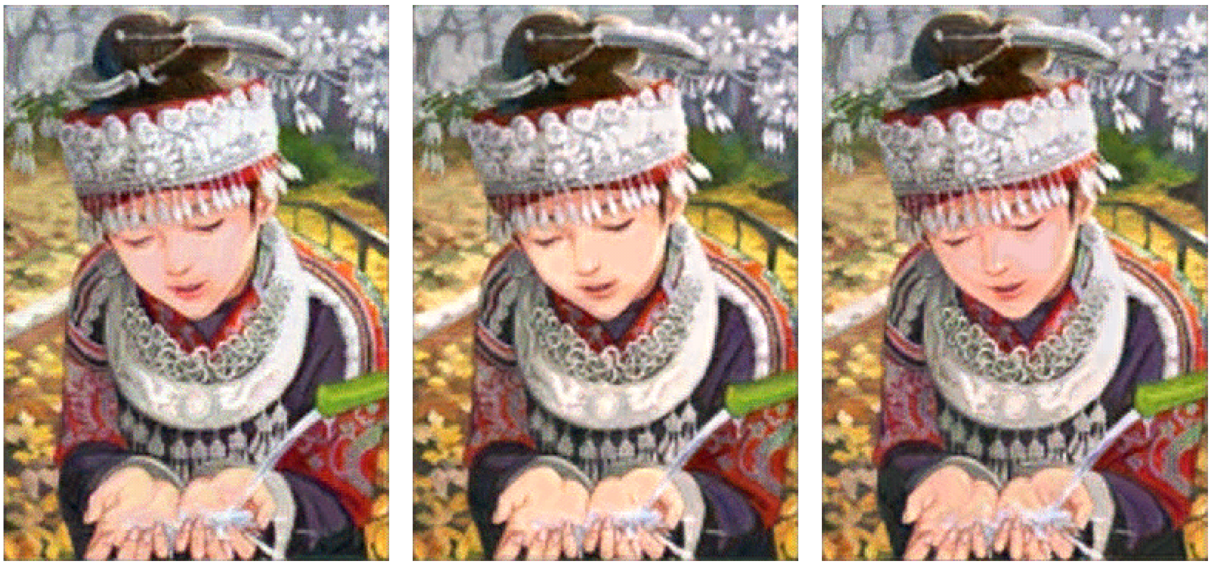


Figure C.9: INT2 MESPCN final images with knowledge distillation

Appendix D

Raw Data Output

D.1 Throughput

`gpu-throughput.json` measuring the average throughput of 32-bit floating point networks on an NVIDIA RTX 3080 (Laptop) GPU.

```json
{
    "ESPCN": {
        "128x128px fps": 2107.6182711010515,
        "256x256px fps": 571.7773519233025,
        "512x512px fps": 146.7033555332811
    },
    "MESPCN": {
        "128x128px fps": 3141.489629063681,
        "256x256px fps": 941.7728239050417,
        "512x512px fps": 239.74339245569794
    },
    "FESPCN": {
        "128x128px fps": 1936.2209211544052,
        "256x256px fps": 547.2519990935104,
        "512x512px fps": 140.28493023038826
    }
}
```

Listing D.1: RTX 3080 (Laptop) throughput per network

D.2 Quantised Distillation Impact on LPIPS

D.2.1 INT8 Networks

α	**MESPCN**
0.05	+2.713%
0.20	+0.838%
0.30	+2.536%
0.80	+5.890%

BSD100

α	**MESPCN**
0.05	+3.625%
0.20	+1.305%
0.30	+2.611%
0.80	+5.705%

Urban100

α	**MESPCN**
0.05	+1.382%
0.20	-0.585%
0.30	-0.266%
0.80	+12.712%

Set5

α	**MESPCN**
0.05	+2.668%
0.20	+1.074%
0.30	+2.342%
0.80	+11.356%

Set14

D.2.2 INT6 Networks

α	**ESPCN**	**MESPCN**
0.05	+0.881%	-0.065%
0.20	+5.835%	+1.316%
0.30	+7.597%	+1.579%
0.80	+9.051%	+6.077%

BSD100

α	**ESPCN**	**MESPCN**
0.05	+1.178%	+0.571%
0.20	+4.431%	+2.071%
0.30	+5.987%	+2.381%
0.80	+8.062%	+6.166%

Urban100

α	**ESPCN**	**MESPCN**
0.05	+3.038%	+1.427%
0.20	+20.481%	+6.150%
0.30	+15.767%	+4.997%
0.80	+20.010%	+19.275%

Set5

α	**ESPCN**	**MESPCN**
0.05	+2.696%	+1.697%
0.20	+9.779%	+3.525%
0.30	+11.046%	+5.189%
0.80	+14.392%	+12.989%

Set14

D.2.3 INT4 Networks

α	**ESPCN**	**MESPCN**
0.05	+6.968%	+1.972%
0.20	+8.318%	+1.204%
0.30	+9.264%	+4.235%
0.80	+8.459%	+7.203%

BSD100

α	**ESPCN**	**MESPCN**
0.05	+5.298%	+1.114%
0.20	+6.240%	+1.382%
0.30	+7.685%	+3.344%
0.80	+8.386%	+5.685%

Urban100

α	**ESPCN**	**MESPCN**
0.05	+21.257%	+7.647%
0.20	+24.636%	+5.469%
0.30	+21.633%	+8.034%
0.80	+27.404%	+17.328%

Set5

α	**ESPCN**	**MESPCN**
0.05	+12.237%	+3.850%
0.20	+11.805%	+2.043%
0.30	+13.158%	+6.161%
0.80	+16.556%	+11.078%

Set14

D.2.4 INT2 Networks

α	**ESPCN**	**MESPCN**
0.05	+3.394%	+1.738%
0.20	+6.500%	-1.924%
0.30	+6.031%	-0.139%
0.80	+6.076%	+1.971%

BSD100

α	**ESPCN**	**MESPCN**
0.05	+3.251%	+0.434%
0.20	+5.869%	-2.084%
0.30	+4.688%	-2.605%
0.80	+5.321%	+1.250%

Urban100

α	**ESPCN**	**MESPCN**
0.05	+4.606%	+2.304%
0.20	+5.415%	-4.318%
0.30	+1.078%	-0.655%
0.80	+6.910%	+2.838%

Set5

α	**ESPCN**	**MESPCN**
0.05	+4.215%	-0.459%
0.20	+5.158%	-3.183%
0.30	+4.566%	-1.379%
0.80	+6.969%	+2.171%

Set14

Appendix E

Codebase

The code for this dissertation can be found at:

`https://github.com/jakedves/image-generation-hardware`