LiU-ITN-TEK-A--20/029--SE

Evaluation of Machine Learning Primitives on a Digital Signal Processor

Vilhelm Engström

2020-06-12

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--20/029--SE

Evaluation of Machine Learning Primitives on a Digital Signal Processor

Examensarbete utfört i Datateknik
vid Tekniska högskolan vid
Linköpings universitet

Vilhelm Engström

Handledare Gabriel Eilertsen
Examinator Aida Nordman

Norrköping 2020-06-12

Abstract

Modern handheld devices rely on specialized hardware for evaluating machine learning algorithms. This thesis investigates the feasibility of using the digital signal processor, a part of the modem of the device, as an alternative to this specialized hardware. Memory management techniques and implementations for evaluating the machine learning primitives convolutional, max-pooling and fully connected layers are proposed. The implementations are evaluated based on to what degree they utilize available hardware units. New instructions for packing data and facilitating instruction pipelining are suggested and evaluated. The results show that convolutional and fully connected layers are well-suited to the processor used. The aptness of the convolutional layer is subject to the kernel being applied with a stride of 1 as larger strides cause the hardware usage to plummet. Max-pooling layers, while not ill-suited, are the most limited in terms of hardware usage. The proposed instructions are shown to have positive effects on the throughput of the implementations.

Acknowledgments

The author would like to thank MediaTek Inc. for the opportunity to work with them and their hardware throughout the thesis. Special thanks to Erik Bertilsson and Henrik Abelsson for providing guidance and feedback during the course of the work. Thanks are extended also to supervisor Gabriel Eilertsen and examiner Aida Nordman for being readily available whenever needed, be it for meetings, feedback, advice or just general discussion.

Contents

List of Figures

List of Tables

1 Introduction

The use of machine learning on handheld devices has seen a significant increase over the past few years. It has become prominent enough that modern smartphones commonly include specialized chips for evaluating machine learning algorithms efficiently [1].

While having a clear use case, adding additional, advanced hardware components may run contrary to other desirable properties, affordability being perhaps chief among them. For this reason, MediaTek has expressed a desire to investigate the feasibility of evaluating machine learning algorithms on their digital signal processor (DSP). The DSP is an already integral part of the modem of most handheld devices. As such, using the DSP for evaluating the algorithms requires no additional hardware.

DSPs are microprocessors specialized in processing digital signals. They are of significant import in smartphones that regularly send and receive large amounts of data over mobile networks [2]. As such, DSPs are vital to the core functionality of most handheld devices whereas machine learning chips are more of an added luxury.

Given the tasks designated to them, DSPs feature a set of properties that differs from that of an ordinary CPU. As a result, they are apt at performing mathematical operations at high speed and with low power consumption [3]. Their high degree of specialization does, however, make them less suited to other tasks. One common limitation that stems from this specialization is that DSPs generally have access to significantly less memory [4] than CPUs.

Considering the large amount of data required by neural networks, it may prove difficult to implement them efficiently on a DSP. If, however, it were to be possible, it would allow for more affordable and powerful handheld devices.

1.1 Motivation

DSPs are designed primarily with the processing of digital signals in mind. The benefits of features such as single-cycle memory access, vector instructions [5] and multiple-access memory [4] are, however, not limited to signal processing. This, in addition to DSPs in handheld devices being largely idle when not transferring data, makes it interesting to investigate how they may be used for solving other problems.

This thesis explores how a DSP may be employed to evaluate convolutional neural networks and whether it is a viable alternative to the specialized AI processing units used in handheld devices today. While it is unlikely that the efficiency of a dedicated AI chip is

reached, it may still be possible to evaluate the algorithms on a sufficiently large DSP within an acceptable amount of time. This would in turn potentially allow for the possibility of forgoing the AI chips in handheld devices and thereby allowing them to be produced at a lower cost. Alternatively, the DSP could be used in tandem with the AI chips in order to increase throughput.

1.2 Aim

The aim of the thesis is to investigate how the machine learning primitives convolutional, max-pooling and fully connected layers may be implemented on a DSP.

A successful approach should utilize inherent strengths of the DSP such as vector instructions and low memory latency. It should also solve issues that make the algorithms less suited to the architecture such as the memory restrictions inherent to the DSP. Additionally, potential changes to the instruction set of the processor that may improve performance should be suggested and evaluated.

1.3 Research Questions

The thesis aims to answer the following research questions:

- How should the large amount of data required by neural networks be processed on a system where the size of both on- and off-chip memory is small?
- What machine learning primitives lend themselves well to vector processing and how, if at all, may the processor be altered to make the fit better?

1.4 Method at a Glance

In order to provide a foundation solid enough for answering the research questions, the thesis proposes algorithms for evaluating convolutional, max-pooling and fully connected layers. The algorithms proposed have been honed by gradually altering them such that they make better use of the available hardware in the DSP provided by MediaTek.

The thesis details ways of managing on- and off-chip memory such that excessive data transfers are avoided and the latency of required data transfers is hidden. It also suggests ways of restructuring certain data in order to increase throughput of the proposed algorithms.

Changes proposed for the processor include two new instructions designed with the implementation of the algorithms in mind. These allow for improved pipelining and packing of data in vector registers using a variable period, respectively. In order to measure the impact of the proposed instructions, they were temporarily added to the instruction set of the DSP.

1.5 Delimitations

For the sake of time-constraints, the thesis considers only convolutional, max-pooling and fully connected layers. The primitives are considered as fully separate rather than as part of an actual network of a particular configuration. As such, the results should be applicable to any convolutional neural network constructed from the primitives in question.

As the aim of the thesis is to investigate the feasibility of running neural networks on a DSP, it is more concerned with the fundamental processes involved than achieving meaningful output of the network. This means that designing a neural network that successfully performs a given task is of less interest. By extension, tuning of network parameters through learning algorithms is left out entirely. Instead, the intent is to investigate different implementations of the chosen primitives and adapt them to the DSP.

The scope of the thesis is limited to work only with the DSP designed and provided by MediaTek and may rely on solutions that are unique to its instruction set. As the latter is not publicly available, any such solution will, when applicable, be described using similar concepts available on more common architectures.

While proposing potential changes that would allow the DSP to more efficiently evaluate the primitives is within the scope of the thesis, these are restricted to deal only with the instruction set. Aspects beyond this such as numeric representations and memory sizes are considered fixed. Furthermore, potential issues that arise due to floating point rounding are not considered.

2 Theory

This chapter presents the theoretical foundation of the thesis. It starts by establishing a mathematical basis and subsequently delves into the inner workings of neural networks. Aspects central to high performance computing such as data parallelism and memory considerations are also presented. The chapter concludes by exploring digital signal processors.

2.1 Mathematical Concepts

The large number of machine learning frameworks available has made it possible to employ convolutional neural networks without much concern for the mathematics behind them. Implementing them, on the other hand, requires an understanding of the underlying primitives. This section begins by describing tensors, a concept used frequently in the context of neural networks. It continues by presenting convolution and cross-correlation, two fundamental building blocks of convolutional neural networks.

2.1.1 Tensors

Tensors are a mathematical generalization of vectors. A tensor has a magnitude and an arbitrary number of directions. The number of these directions for a particular tensor is referred to as its order. By this definition, a tensor of order 0 has a magnitude and no direction associated with it. In other words, a tensor of order 0 is a scalar. Tensors of order 1 have a magnitude and a single direction, meaning they are vectors [6].

Higher order tensors are less intuitive. A tensor of order 2, a so-called dyad, is constructed by the dyad product of two vectors. Given two vectors $\mathbf{u} = u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}$ and $\mathbf{v} = v_1\mathbf{e_1} + v_2\mathbf{e_2} + v_3\mathbf{e_3}$, the dyad product $\mathbf{uv}$ is given by

$$\mathbf{uv} = \sum_{i=1}^{3}\sum_{j=1}^{3} u_i v_j \mathbf{e}_i\mathbf{e}_j \tag{2.1}$$

where $\mathbf{e_1}$, $\mathbf{e_2}$ and $\mathbf{e_3}$ are linearly independent unit vectors [6].

Second order tensors have a single magnitude and two directions. By allowing the subscripts i and j in (2.1) to denote row and column, respectively, the scalar components of the dyad can be arranged in a matrix [6] as shown in Fig. 2.1.

$$\begin{pmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \end{pmatrix}$$

Figure 2.1: Scalar components of a second order tensor visualized in a matrix-like structure.

Order three tensors are defined by the triad product **uvw** [6] where **u** and **v** are defined as above and $\mathbf{w} = w_1\mathbf{e}_1 + w_2\mathbf{e}_2 + w_3\mathbf{e}_3$. The triad product is computed in a similar way to its dyad counterpart but sums over an additional dimension. More specifically,

$$\mathbf{uvw} = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} u_i v_j w_k \mathbf{e}_i\mathbf{e}_j\mathbf{e}_k \tag{2.2}$$

computes the triad product **uvw** [6].

Tensors of an arbitrary order $n \in \mathbb{Z}^+, n > 3$ are defined analogously to their lower order counterparts and, similarly, have n directions and 3^n components [6].

In computer science, the term tensor is generally used as a synonym for the scalar components of the mathematical tensor. As such, they are no more than multi-dimensional arrays and are used in order to generalize matrices beyond two dimensions [7]. Additionally, the restriction of the tensor containing 3^n components is lifted. As a consequence, a tensor of order three may be used to reference any arbitrary, three-dimensional array [8].

When used in the thesis, unless otherwise stated, the term tensor refers to the more liberal computer science interpretation.

2.1.2 Convolution

Convolution is a mathematical operation that given two functions produces a third [9]. For two continuous signals $f(t)$ and $g(t)$,

$$h(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.3}$$

computes their convolution $h(t)$ [10].

In fields such as image processing, machine learning and signal processing, the data processed is discrete. As a result, researchers in these fields typically present convolution as a summation over discrete points. Considering two discrete functions $f'(n)$ and $g'(n)$,

$$h'(n) = (f' * g')(n) = \sum_{i=-\infty}^{\infty} f'(i)g'(n-i) \tag{2.4}$$

yields their discrete convolution $h'(n)$. As can be seen, discrete convolution computes the inner products of local neighborhoods of signal samples and a time-reversed kernel [10]. The fact that the kernel is time-reversed is indicated by the sign preceding i in $g'(n-i)$. In (2.4), the input signal is $g'(n)$, the kernel $f'(n)$ and the convolution is performed over the neighborhood of the n^{th} sample.

In practical applications such as image processing and machine learning, the convolution kernel is defined to be non-zero over a finite number of data points. As a consequence, the infinite summation interval can be reduced to comprise a finite set of elements [8].

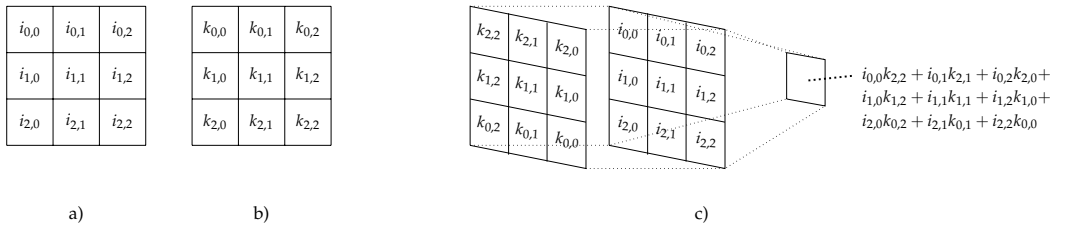


Figure 2.2: Application of a 3×3 convolution filter to a 3×3 image. a) The input image. b) The filter kernel. c) Application of the filter kernel to the input image in order to produce a convolved feature.

Convolution may be extended to higher dimensions. The discrete convolution of a function $g^{2d}(n_x, n_y)$ and a two-dimensional kernel $f^{2d}(n_x, n_y)$ is computed using

$$h^{2d}(n_x, n_y) = (f^{2d} * g^{2d})(n_x, n_y) = \sum_i \sum_j f^{2d}(i, j) g^{2d}(n_x - i, n_y - j) \tag{2.5}$$

where $h^{2d}(n_x, n_y)$ is the so-called convolved feature [8] and n_x and n_y denote indices for column and row in the image, respectively. Additionally, the boundaries for i and j are left out as they are of limited importance in practice. Fig. 2.2 shows how the two-dimensional filter kernel is reversed when convolving an image.

Example usages of two-dimensional convolution include noise reduction and edge detection in image processing [10]. It is, however, limited to greyscale images. In order to convolve color images, three-dimensional convolution is required. For two given functions $f^{3d}(n_x, n_y, n_z)$ and $g^{3d}(n_x, n_y, n_z)$, their convolved feature $h^{3d}(n_x, n_y, n_z)$ is computed as

$$h^{3d} = (f^{3d} * g^{3d}) = \sum_i \sum_j \sum_k f^{3d}(i, j, k) g^{3d}(n_x - i, n_y - j, n_z - k). \tag{2.6}$$

Here, the fact that h^{3d} and $(f^{3d} * g^{3d})$ are both functions of n_x, n_y and n_z is left out for notational convenience.

2.1.2.1 Fourier Transform-Based Convolution

Whereas convolution may be computed by simply applying the mathematical definition, other methods have been explored. One such method relies on convolution in the time domain transforming to multiplication in the frequency domain. Instead of convolving the input in time domain, both the signal and the kernel are transformed using a fast Fourier transform. This is followed by a multiplication and an inverse Fourier transform, yielding the convolved feature of the input [11]. More specifically, the Fourier transform-based convolution of two functions f and g is computed using

$$(f * g)(t) = \mathcal{F}^{-1}(\mathcal{F}(f(t))\mathcal{F}(g(t))) \tag{2.7}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote Fourier transform and inverse Fourier transform, respectively.

In terms of performance, Fourier transform-based convolution may outperform direct convolution under certain conditions. Assuming an $n \times n$ input image and a $k \times k$ filter kernel, direct, two-dimensional convolution requires $(n - k + 1)^2 k^2 \in O(n^2 k^2)$ multiplications. The number of multiplications in Fourier transform-based convolution is $cn^2 \log(n) + 4n^2 \in O(n^2 \log(n))$ for some $c \in \mathbb{R}$ [11].

2.1.2.2 Separable Convolution

Another common convolution approach is so-called separable convolution. The technique separates the filter kernel into two separate components, one vertical and one horizontal. The

actual convolution is performed over two steps. In the first step, the input is convolved with the first of the separated components, producing an intermediary signal. This intermediary is then convolved with the second separated component, producing the convolved feature [3].

A kernel being separable means that it can be decomposed into a horizontal and a vertical part that when convolved with each other produce the original kernel [3]. Using the horizontal Sobel operator as an example, this can be summarized as

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{2.8}$$

where the vectors in the right-hand side are the separated components and $*$ is the convolution operator.

Separable two-dimensional convolution of an $n \times n$ size image with a $k \times k$ kernel is computed in time $O(n^2k)$ [3]. As such, it is strictly less computationally expensive than direct convolution. The difficulty with the approach lies in ascertaining that a particular convolution kernel is separable.

Whether a two-dimensional kernel is separable can be determined by computing the rank of the corresponding matrix. If the matrix is of rank 1, meaning all but one of its columns are linearly dependent, the kernel is separable [12]. Separability of tensors of arbitrary order is instead determined via the so-called nuclear norm. As with the rank of a matrix, a tensor is separable if and only if its nuclear norm is 1 [13]. Whereas matrix rank can be computed in polynomial time using Gaussian elimination [14], determining the nuclear norm of even an order-three tensor is NP-hard [15].

2.1.3 Cross-Correlation

The cross-correlation $c(t)$ of two continuous functions is given by

$$c(t) = (f \star g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t + \tau) d\tau \tag{2.9}$$

where $f(t)$ is the filter kernel and $g(t)$ the input signal [16]. Comparing this to (2.3), it is evident that the only difference between convolution and cross-correlation is the sign preceding the τ in the second term of the integral [8].

Cross-correlation can be both discretized and extended to higher dimensions in the same way as convolution [8]. Consequently, given two discrete functions $f^{3d}(n_x, n_y, n_z)$ and $g^{3d}(n_x, n_y, n_z)$, their cross-correlation $c^{3d}(n_x, n_y, n_z)$ is given by

$$c^{3d} = (f^{3d} \star g^{3d}) = \sum_i \sum_j \sum_k f^{3d}(i, j, k) g^{3d}(n_x + i, n_y + j, n_z + k). \tag{2.10}$$

Like (2.6), (2.10) does not explicitly specify that both c^{3d} and $(f^{3d} \star g^{3d})$ are functions of n_x, n_y and n_z for notational convenience. As can be seen, the only difference between (2.6) and (2.10) is that the subtractions in (2.6) are replaced by additions in (2.10).

Conceptually, replacing the subtractions with additions means that the filter kernel is no longer spatially reversed when computing the inner products [8]. Fig. 2.3 illustrates this. The main difference between convolution and cross-correlation in terms of mathematical properties is that the former is commutative whereas the latter is not [8].

2.2 Neural Networks

Neural networks are among the most popular machine learning models used today [8]. They are commonly visualized as directed acyclic graphs consisting of an input layer, one or more

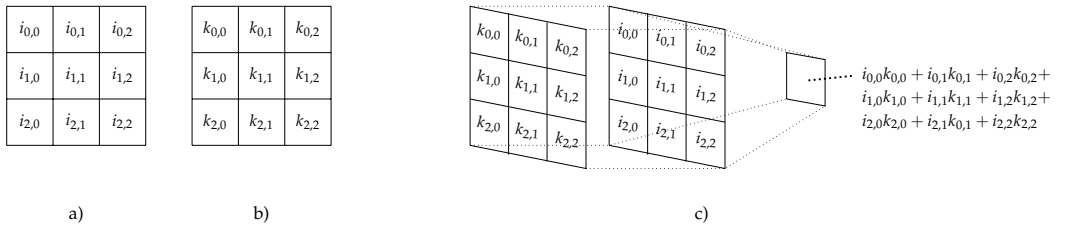


Figure 2.3: Application of a 3×3 cross-correlation filter to a 3×3 image. a) The input image. b) The filter kernel. c) Application of the filter kernel to the input image and the resulting output.

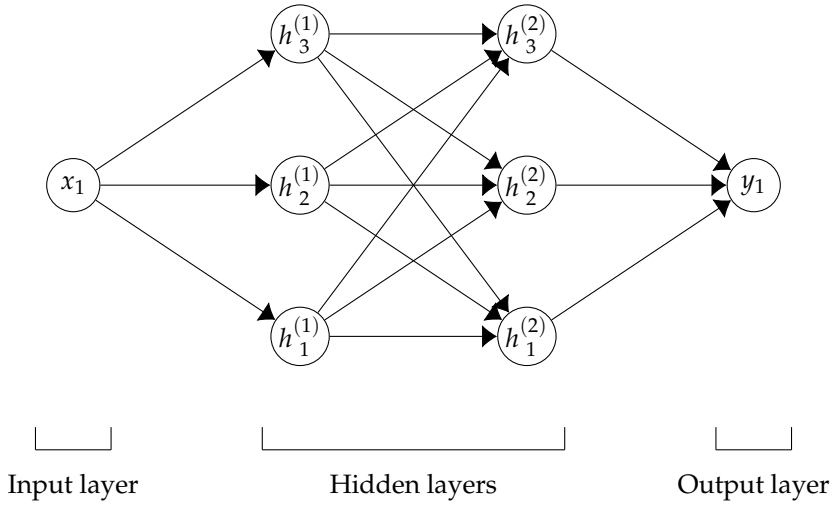


Figure 2.4: A feedforward neural network with a single input, a single output and two hidden layers, each of which consisting of three neurons. The superscripts of the hidden neurons indicate to which layer they belong and the subscripts their index in said layer. Neither weights nor biases are shown.

hidden layers and an output layer. Formally, the type of network that is usually shown is called a feedforward neural network [8]. Fig. 2.4 shows such a network containing two hidden layers.

As can be seen in Fig. 2.4, each neuron except for the input x_1 is connected to all neurons in the preceding layer. Generally, when each neuron in the n^{th} layer $L^{(n)}$ is connected to all the neurons in its input layer $L^{(n-1)}$, $L^{(n)}$ is said to be fully connected. A fundamental property of feedforward networks is that all layers but the input layer fulfill this property [8].

Fig. 2.4 omits a few important aspects, namely weights, biases and hidden units. The weights of a feedforward neural network correspond to the edges of the graph representation [17]. Fig. 2.5 shows the input and first hidden layer of the network in Fig. 2.4 with the weights added. The weights perform a per-neuron scaling of the input according to

$$a_j^{(n)} = \sum_{i=1}^{d} w_{i,j}^{(n)} h_i^{(n-1)} \tag{2.11}$$

where $a_j^{(n)} \in \mathbb{R}$ is a linear combination of the inputs of neuron j in layer $L^{(n)}$. The upper limit $d \in \mathbb{Z}^+$ is the number of inputs of said layer and the weight $w_{i,j}^{(n)} \in \mathbb{R}$ is associated with the

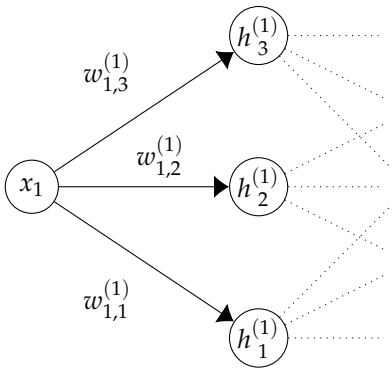


Figure 2.5: Input and first hidden layer of the network shown in Fig. 2.4 with added weights. The superscripts of the weights denote which layer the neuron at the end of the edge belongs to. The subscripts indicate which neurons in the two layers are connected.

edge connecting neuron i in layer $L^{(n-1)}$ with neuron j in layer $L^{(n)}$. The so-called activation $h_i^{(n-1)} \in \mathbb{R}$ of the i^{th} neuron in layer $L^{(n-1)}$ serves as the i^{th} input to layer $L^{(n)}$ [17].

For the part of the network that is shown in Fig. 2.5, there is a single input x_1, meaning $d = 1$ and the summation in (2.11) is performed over a single element. Using this, the linear combinations $a_1^{(1)}$, $a_2^{(1)}$ and $a_3^{(1)}$ corresponding to neurons $h_1^{(1)}$, $h_2^{(1)}$ and $h_3^{(1)}$ are given by

$$a_1^{(1)} = w_{1,1}^{(1)} x_1, \tag{2.12}$$

$$a_2^{(1)} = w_{1,2}^{(1)} x_1, \tag{2.13}$$

and

$$a_3^{(1)} = w_{1,3}^{(1)} x_1 \tag{2.14}$$

respectively. As can be seen, the linear combinations are obtained by scaling the input of the respective neurons with its corresponding weight. This model can be further refined by adding a bias for each neuron [17]. The result of adding a bias for each neuron in Fig. 2.5 is shown in Fig. 2.6.

With the added bias terms $w_{0,j}^{(1)} \in \mathbb{R}$, the linear combination $a_j^{(1)} \in \mathbb{R}$, $j = 1, 2, 3$ of neuron $h_j^{(1)}$ in Fig. 2.6 is computed via

$$a_j^{(1)} = w_{1,j}^{(1)} x_i + w_{0,j}^{(1)}. \tag{2.15}$$

This is generalized to compute the linear combination $a_j^{(n)}$ corresponding to the j^{th} neuron in layer $L^{(n)}$ using

$$a_j^{(n)} = \sum_{i=1}^{d} w_{i,j}^{(n)} h_i^{(n-1)} + w_{0,j}^{(n)} \tag{2.16}$$

where d is the number of layer inputs and $h_i^{(n-1)}$ is the activation of the i^{th} neuron of layer $L^{(n-1)}$.

The activation $h_j^{(n)}$ of the j^{th} neuron in layer $L^{(n)}$ is produced by feeding its corresponding linear combination $a_j^{(n)}$ through a so-called activation function. The latter is a non-linear, piecewise differentiable function. Common choices include the sigmoid [17] defined as

$$\sigma(x) = \frac{1}{1 + e^x} \tag{2.17}$$

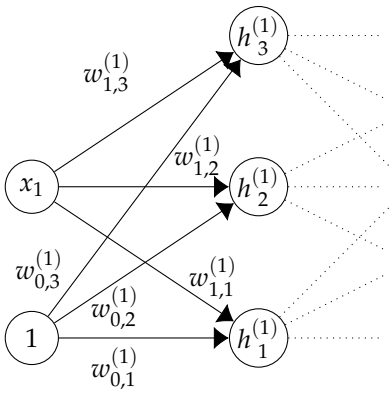


Figure 2.6: Fig. 2.5 with a bias $w_{0,j}^{(1)}$ added for each hidden neuron $h_j^{(1)}$, $j = 1, 2, 3$. The biases are visualized as weights for edges between neurons in the hidden layer and an input neuron with value 1. When evaluating the network, the contribution of this input neuron is its value, 1, multiplied with its weight, the bias $w_{0,j}^{(1)}$, meaning that the contribution of the bias is $1 \cdot w_{0,j}^{(1)} = w_{0,j}^{(1)}$.

and the rectified linear unit, known as the ReLU [8], given by

$$\text{ReLU}(x) = \max(0, x). \tag{2.18}$$

The output $h_j^{(n)}$ of the activation function for the j^{th} hidden neuron in the n^{th} layer $L^{(n)}$ is used as input for the neurons in the subsequent layer $L^{(n+1)}$ and the same procedure is repeated [17]. In other words, the activation $h_j^{(n)} \in \mathbb{R}$ of neuron j in layer $L^{(n)}$ is computed using

$$h_j^{(n)} = \Phi\left(a_j^{(n)}\right) = \Phi\left(\sum_{i=1}^{d} w_{i,j}^{(n)} h_j^{(n-1)} + w_{0,j}^{(n)}\right) \tag{2.19}$$

where $\Phi : \mathbb{R} \to \mathbb{R}$ is a general activation function and $w_{i,j}^{(n)}$ is the weight applied to the i^{th} layer input $h_i^{(n-1)}$. The term $w_{0,j}^{(n)}$ is the bias corresponding to the j^{th} neuron in layer $L^{(n)}$ and d is the number of layer inputs.

When the data has propagated through the entire network and reached the output neuron, the input of the latter is transformed to a meaningful output. This meaningful output may, as an example, be a vector of class scores. The transformation is performed using yet another activation function [17]. This activation function is usually different from the ones used in earlier layers and is chosen according to what fits the particular task of the network [18].

Fig. 2.7 shows a simple feedforward network complete with weights and biases for each neuron. Choosing the ReLU and sigmoid functions as the activation functions for the hidden and output layers respectively, the entire forward pass of the network in Fig. 2.7 is given by

$$y_1 = \sigma\left(\sum_{j=1}^{2} w_{j,1}^{(2)} \text{ReLU}\left(x_1 w_{1,j}^{(1)} + w_{0,j}^{(1)}\right) + w_{0,1}^{(2)}\right). \tag{2.20}$$

Here, the input $x_1 \in \mathbb{R}$ is first scaled with a weight $w_{1,j}^{(1)} \in \mathbb{R}$ in the first layer and added with the corresponding bias $w_{0,j}^{(1)} \in \mathbb{R}$. The sum is passed through the ReLU activation function. This is done for each neuron $h_j^{(1)}$ in the hidden layer, hence the summation over j. The computed values, the so-called output unit activations, effectively serve as the input for the

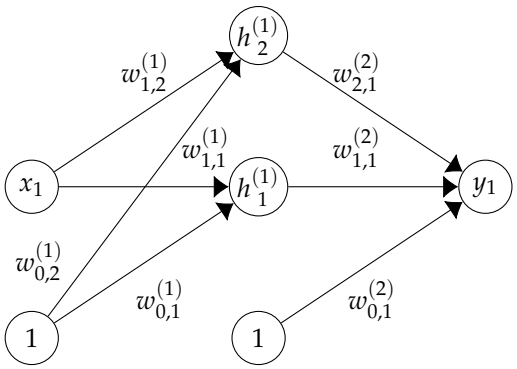


Figure 2.7: A feedforward network with weights $w_{i,j}^{(n)}$, $i, j = 1, 2, 3$ and biases $w_{0,j}^{(n)}$ shown for each neuron.

output layer. As such, they are scaled with their corresponding weights $w_{j,1}^{(2)} \in \mathbb{R}$. The sum of the scaled output unit activations is then added with the output bias $w_{0,1}^{(2)} \in \mathbb{R}$ and, finally, this sum is passed through the sigmoid activation function to produce the output $y_1 \in \mathbb{R}$ of the network.

2.2.1 Neural Network Pruning

As the applications of neural networks become increasingly complex, so do the networks themselves [19]. Larger networks require a larger number of computations to evaluate. This makes using neural networks on resource-constrained systems such as embedded devices difficult [20].

Neural network pruning is an umbrella term for different techniques used to reduce the computational complexity of neural networks [21]. This is usually achieved by removing individual weights of the network based on certain criteria. Which weights to remove is determined by including a measure of the computational complexity in the cost function used when training the network [19].

As pruning reduces the total number of weights of the network, it may affect how well said network manages the tasks designated to it [19]. While it might initially seem as though reducing the number of weights in the network would lead to less accurate results, this is not necessarily the case. Instead, reducing the number of weights can improve how well a network generalizes to arbitrary inputs. Pruning should, however, be used with moderation as removing too many weights will result in the network no longer being able to approximate the data [19].

2.3 Convolutional Neural Networks

While feedforward networks are powerful, their high degree of connectivity makes them less suited to certain tasks. Connecting each neuron in a layer to each neuron in the prior means that all inputs are considered at each position, something that is not necessarily desirable. A common scenario when this is less suitable is when working with images as pixels close to each other tend to have a higher correlation than pixels farther apart [17]. For these types of input, convolutional neural networks are usually preferred instead.

As image processing is the perhaps most common application of convolutional networks, this section assumes the network input is a three-channel color image. The principles do, however, apply to any three-dimensional type of data.

Whereas feedforward networks accept an arbitrary number of scalar inputs, convolutional networks are usually considered taking a single order-three tensor. When working with im-

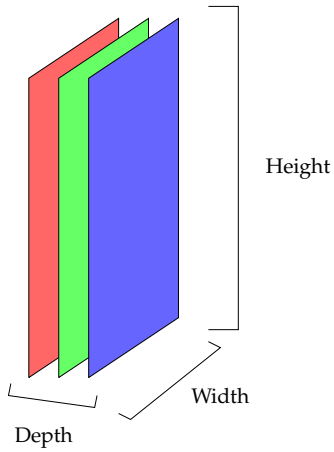


Figure 2.8: A three-channel color image as seen by a convolutional neural network.

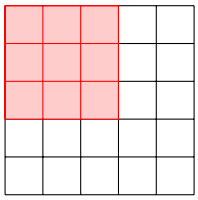


Figure 2.9: The receptive field of a convolutional network. The black grid is a single channel of the input image, the red area shows the receptive field, in this case a 3×3 neighborhood.

ages, the conceptual width and height of the tensor correspond to the width and height of the input image. The three depth channels of the tensor represent the red, green and blue channels, respectively [22]. These depth channels are commonly referred to as feature maps [17]. A possible input of a convolutional network is shown in Fig. 2.8. The feature maps of the input correspond to the red, green and blue planes in the illustration.

A key concept of convolutional networks is that they work with so-called local receptive fields [17], meaning that only part of the input image is processed at a time. In practice, this is realized by applying a smaller kernel to the image over a series of steps. Fig. 2.9 shows the receptive field of a network using a 3×3 kernel. The spatial locality of the kernel allows for extracting local features from subregions of the image. These local features may in subsequent layers be combined with others, ultimately yielding information about the image as a whole [17]. Fig. 2.10 shows how two subregions processed separately may in a later layer both come to influence the result.

As the input is processed in smaller parts at a time, convolutional networks allow for what is called weight sharing. This means that rather than storing a different weight for each pair of connected neurons as done in feedforward networks, a single, smaller kernel suffices. This kernel consists of the layer weights and is typically shared for the entire layer [17]. For large neural networks, this weight sharing may allow for significantly reduced memory usage. Additionally, it contributes to the so-called translation invariance of the network [23].

Sharing the kernel of weights among all neurons in a layer means that the evaluation of the activation of a neuron is equivalent to convolving its input with the kernel [17]. This is, however, rarely how convolutional layers in convolutional networks are implemented. Instead, a large number of machine learning frameworks prefer using cross-correlation. This yields the same observable behavior of the network despite the mathematical result being different [8].

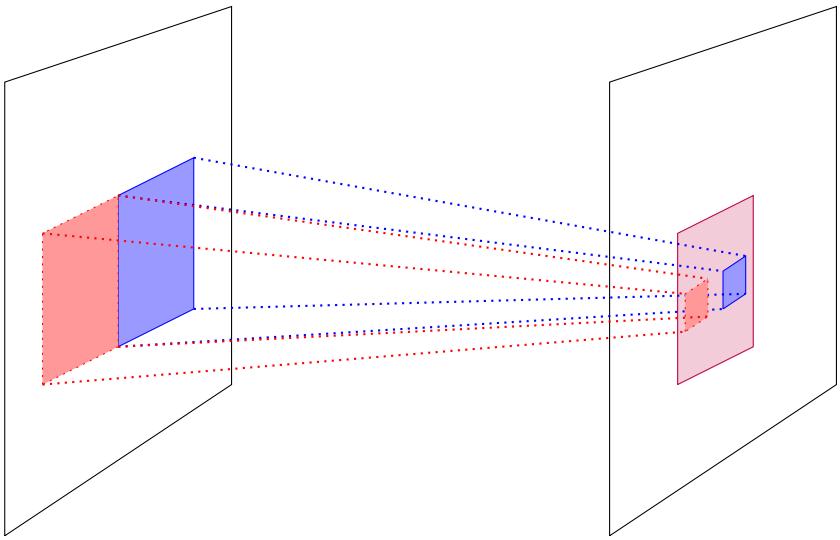


Figure 2.10: Combination of local features in a later layer. The red and blue kernels in the left layer process different areas of the image. The output of these separate operations are both processed as part of the purple kernel in the right layer. For simplicity, only one depth channel is shown.

The activations produced by the convolutional layers are, like in feedforward networks, fed through a non-linear activation function before they are used as input for the subsequent layer. The ReLU is a common choice of activation function here as well [8].

In addition to convolutional layers with added non-linearities, downsampling is an important property employed by convolutional networks. The downsampling is more commonly referred to as pooling and it replaces an area of its input with a summary statistic of said area [8]. A common choice of summary is to simply choose the maximum intensity of the area. This particular version of pooling is referred to as max-pooling [22].

In some literature, the convolution, activation and pooling are considered to constitute a single, three-part layer of the network. Others choose to treat pooling as a separate layer. In this thesis, the latter is preferred as notable convolutional networks such as the VGG16 consist of several convolutional layers with no pooling layers in-between [24].

The final primitive to be considered in the thesis is the fully connected layer. This layer is reminiscent of the ones in feedforward networks in the sense that each neuron is connected to all neurons in its input layer [22]. As fully connected layers are both compute- and memory-intensive, they are generally used sparingly and among the last layers of a network. Examples of this include the VGG16 [24] and AlexNet [25].

2.3.1 Hyperparameters

Processing the image in parts introduces a need for the so-called hyperparameters stride, padding and output depth. Stride refers to with what interval the kernel is applied to the image whereas padding is a means of preserving information at the borders of the image [22]. The output depth determines the number of depth channels in the subsequent layer [26].

The stride controls the overlap of the kernels and, by extension, the size of the output [22]. Fig. 2.11 shows the procedure of applying a 3×3 kernel to a 6×6 single-channel image with a stride of 1. At each stage, the kernel is moved a single step to the right. When the end of a row is reached, the kernel is moved one step down to the next. In each step, a single value is produced for a total of 16.

Fig. 2.12 shows the result of applying a 3×3 kernel to a 6×6 single-channel image with a stride of 3. Here, the kernel is moved 3 pixels along each row at every step. When the end

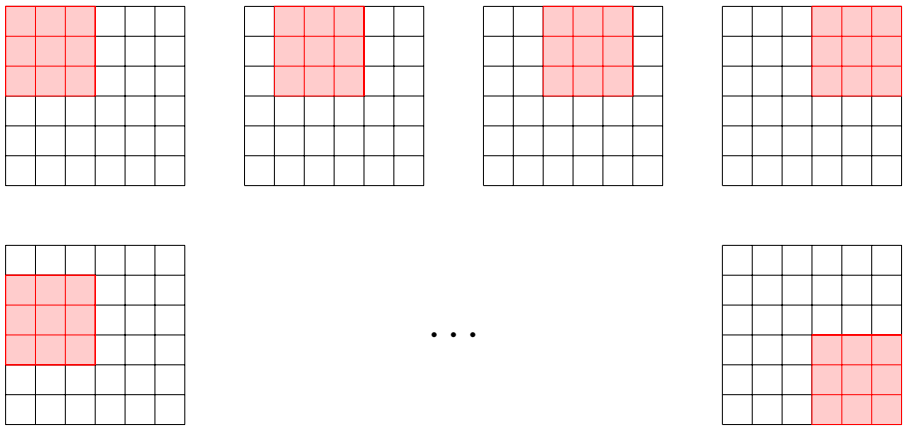


Figure 2.11: Applying a 3×3 kernel to a 6×6 image with a stride of 1.

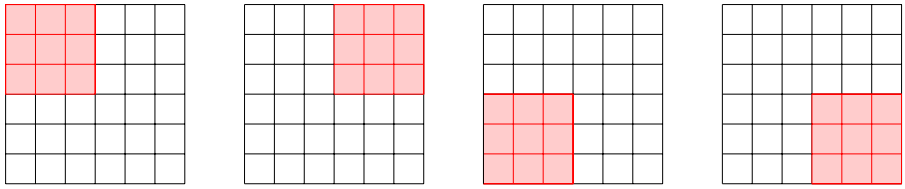


Figure 2.12: Applying 3×3 kernel to a 6×6 image with a stride of 3.

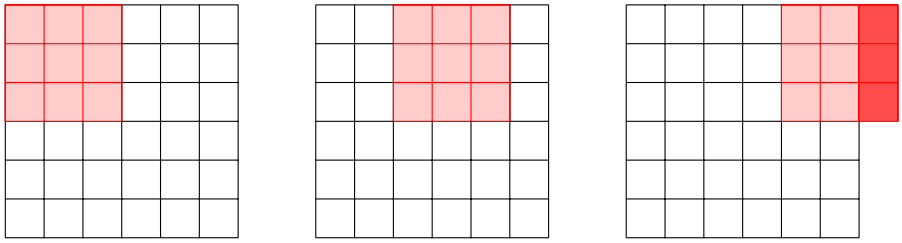


Figure 2.13: An attempt at applying a 3×3 kernel to a 6×6 image with a stride of 2. The choice of hyperparameters does not yield an integer when computing (2.21), meaning the kernel will overstep. The part that falls outside the image is shown in brighter red.

of a row is reached, the kernel is moved 3 rows down. Since the kernel is applied at only 4 positions, the filtering produces a total of 4 values. As is evident, a larger stride results in a smaller output. Assuming a $k \times k$ size kernel, an $n \times n$ image and a stride $s \in \mathbb{Z}^+$,

$$o = \frac{n-k}{s} + 1 \tag{2.21}$$

computes the size $o \in \mathbb{Z}^+$ of the square output [22].

As the width and height of an image are expressed in pixels, it can be observed that the fraction in (2.21) must yield an integer. This means that the stride s should be chosen such that it divides $n - k$. The result of not respecting this constraint is shown in Fig. 2.13.

As seen in Fig. 2.11, kernel applications may cause the spatial size of the output to be smaller than that of the input. This is not always desirable, especially not when several convolutional layers using the same size filter kernel are used in direct succession. Fig. 2.14 aims to demonstrate this issue more clearly. It shows the result of convolving a 5×5 image with a 3×3 kernel using a stride of 2. As can be seen, the convolved feature has the dimensions 2×2, meaning there is no way to apply the same 3×3 filter kernel to it.

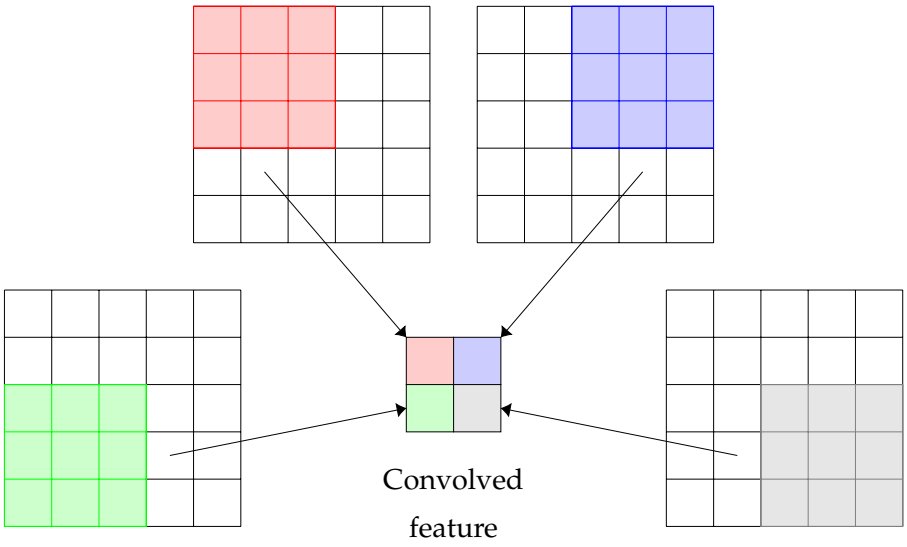


Figure 2.14: A 3×3 kernel applied to a 5×5 image with stride 2. The 2×2 convolved feature is shown in the center. The colors indicate which application of the filter yielded which part of the convolved feature.

0	0	0	0	0	0	0
0						0
0						0
0						0
0						0
0						0
0	0	0	0	0	0	0

Figure 2.15: A 5×5 image with a zero padding of 1. The shaded area indicates the original image whereas the cells with zeroes are added.

The downsizing that occurs during a filter application can be regarded as an unwanted loss of information. The most common way of addressing the issue is by zero-padding the input. This means that extra rows and columns containing all zeroes are added to the input image [22]. An example of this can be seen in Fig. 2.15 where a 5×5 image with a padding of 1 is shown. While zeroes are the most common form of padding, alternative approaches such as repeating [27] or using mean values of [28] edge pixels have been proposed.

Padding allows for controlling the spatial size of the convolved feature [26]. Fig. 2.16 shows how using the same image, kernel and stride as in Fig. 2.14 but with a zero padding of 1 impacts the convolved feature. Instead of the output having dimensions 2×2 as in Fig. 2.14, the convolved feature in Fig. 2.16 is 3×3. The size of the output can be computed by including the padding in (2.21). With k and s again corresponding to the kernel size and stride, respectively,

$$o = \frac{n - k + 2p}{s} + 1 \tag{2.22}$$

computes the size o of the convolved feature obtained when convolving an $n \times n$ input image with padding $p \in \mathbb{Z}_0^+$ [22].

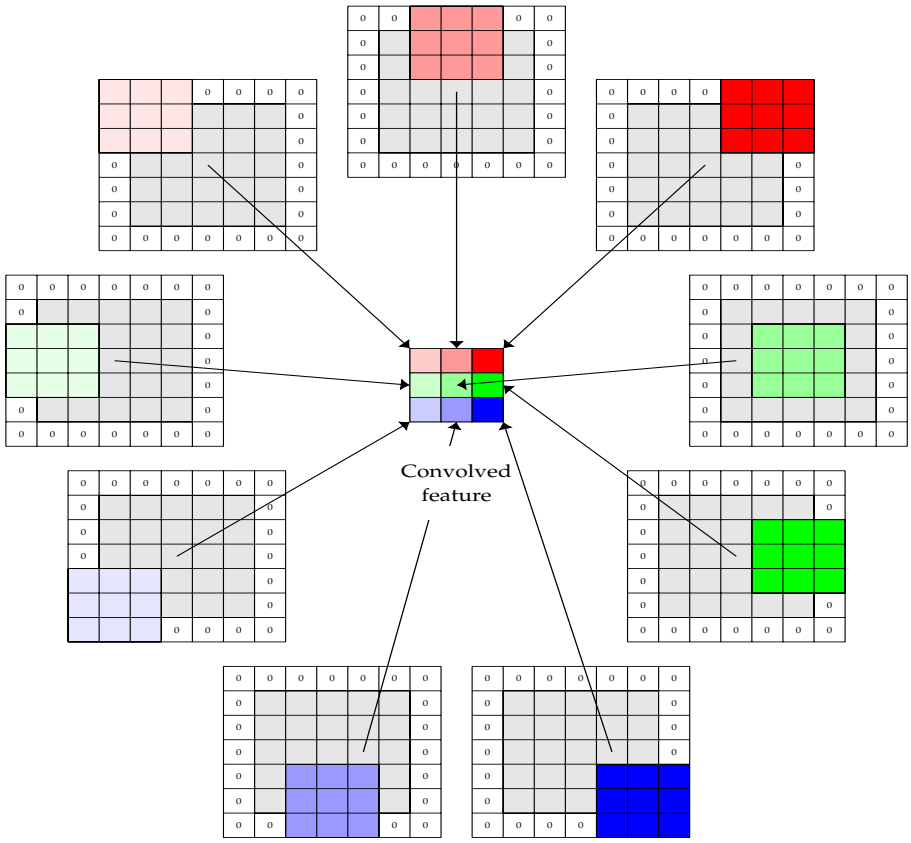


Figure 2.16: Convolving a padded 5×5 image using a 3×3 kernel applied with stride 2. The shaded gray area indicates the original image, the cells with zeroes are the padding.

Whereas padding allows for manipulating the width and height of the output, it does not change the output depth. This is instead controlled by the number of neurons used to look at the same area of the image [26]. In practice, this equates to applying several kernels to the same position of the image and allowing each to contribute to a separate depth channel [29]. This is illustrated in Fig. 2.17.

2.3.2 Convolutional Layer

The convolutional layers are referred to as the feature detectors of a convolutional network [26]. As the appellation implies, they are used to identify feature representations in the input [30]. What these features are depends on how the network is trained, but common examples are edges and other intensity changes in the image [26].

When working with three-channel color images, the main operation performed by the convolutional layers is a three-dimensional convolution [29] or, in practice, cross-correlation [8]. Despite the mathematical differences, this section does not distinguish between the two as they, for the purpose of neural networks, function identically [8]. As a result, the illustrations in this section show cross-correlation rather than proper convolution.

As the convolution is computed over three dimensions, both the input image and the kernel are, in the general case, three-dimensional [29]. Like two-dimensional convolution, three-dimensional convolution is performed by sliding a kernel over the image. The main difference is the added depth, meaning that a convolution of a depth-three kernel and a depth-three image effectively performs three two-dimensional convolutions, one for each depth channel [31]. The contributions of the three channels are summed up, a bias added

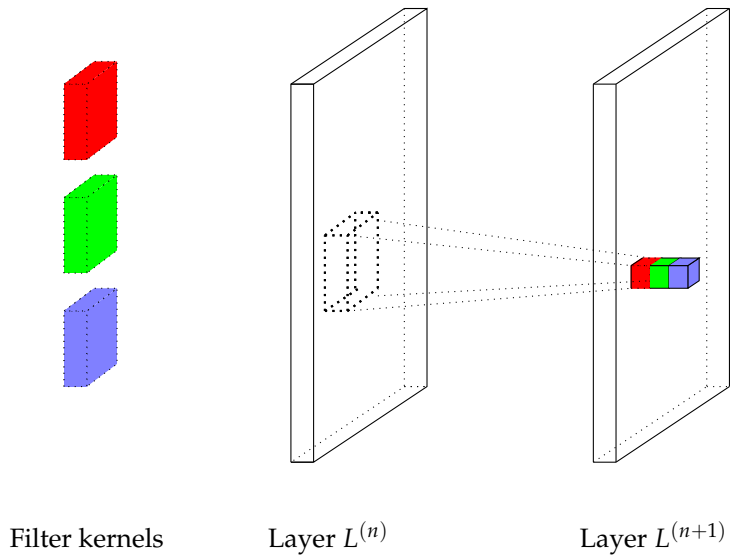


Figure 2.17: Several three-dimensional filter kernels used to produce multiple depth channels in layer $L^{(n+1)}$. The dotted outline in layer $L^{(n)}$ is the position to which each of the kernels is to be applied. The red, green and blue kernels are applied separately and the output of each application is used for a separate depth channel in layer $L^{(n+1)}$.

and the result passed through a non-linear activation function [30] such as the ReLU. The non-linearity introduced by the activation function is what allows the network to "learn" [8]. More formally, the non-linearity allows the network to approximate any arbitrary mathematical function [32]. The output of the activation function is used as the contribution to the convolved feature [30]. The entire procedure for a single spatial position is shown in Fig. 2.18. Here, the convolution of each depth channel is shown as a separate two-dimensional convolution.

Despite convolutional layers theoretically performing a three-dimensional convolution, the feature maps of the input are typically treated separately [33]. This is sometimes referred to as depthwise convolution wherein independent, two-dimensional convolution is applied to each depth channel [34]. As a result, computing the activation of neuron $h^{(n)}_{i,j,k_n} \in \mathbb{R}$ at position (i,j) in depth channel $k_n \in \mathbb{Z}^+$, $k_n \leqslant d_n$ of layer $L^{(n)}$ can be summarized as

$$h^{(n)}_{i,j,k_n} = \Phi\left(\sum_{k_{n-1}} \left(f^{(n)}_{k_n} * h^{(n-1)}_{k_{n-1}}\right)_{i,j} + b^{(n)}_{i,j,k_n}\right) \tag{2.23}$$

where $\Phi : \mathbb{R} \to \mathbb{R}$ denotes an arbitrary activation function and $b^{(n)}_{i,j,k_n} \in \mathbb{R}$ the bias for position (i,j) of depth channel k_n. The term $(f^{(n)}_{k_n} * h^{(n-1)}_{k_{n-1}})_{i,j}$ is the value at position (i,j) of the convolved feature resulting from the two-dimensional convolution of filter kernel $f^{(n)}_{k_n}$ and the activation $h^{(n-1)}_{k_{n-1}}$ of layer $L^{(n-1)}$. The summation is performed over the respective depth channels $k_{n-1} \in \mathbb{Z}^+, k_{n-1} \leqslant d_{n-1}$ of layer $L^{(n-1)}$. The layer transforms from $d_{n-1} \in \mathbb{Z}^+$ depth channels in layer $L^{(n-1)}$ to $d_n \in \mathbb{Z}^+$ channels in layer $L^{(n)}$.

The number of feature maps produced by a convolutional layer is determined by the number of kernels used [29]. This is not necessarily chosen to keep the number of depth channels constant between layers. As an example, the first convolutional layer in the well-known AlexNet scales the number of depth channels from 3 to 48 by applying 48 $11 \times 11 \times 3$

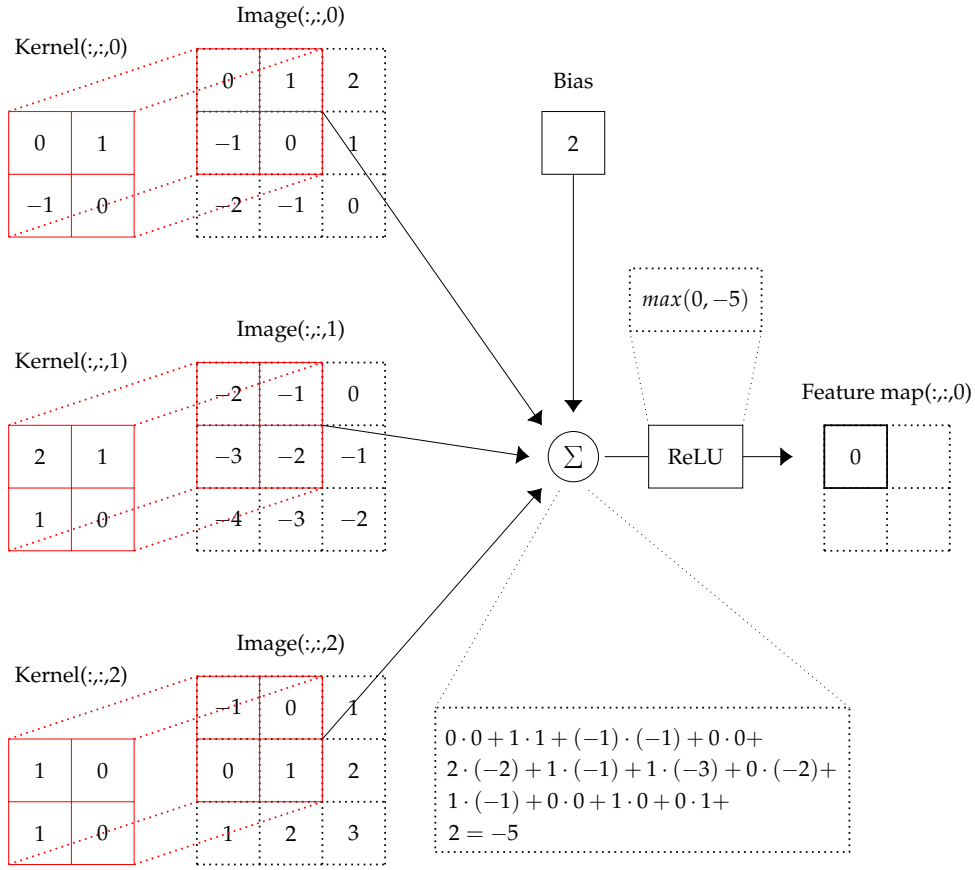


Figure 2.18: Three-dimensional cross-correlation with added bias. ReLU is used as the activation function. The notation $A(:,:,n)$ signifies depth channel $n \in \mathbb{Z}_0^+, n \leqslant 2$ of order-three tensor A. The kernel is slid over the image and at each position the cross-correlation between the kernel and the image is computed. This is done in a per-channel fashion, meaning kernel depth channel 0 is applied to image depth channel 0, kernel channel 1 is applied to image channel 1 and so on. The sum of the contributions of the depth channels is added with the bias. This sum is passed through the ReLU activation function and the output placed in the feature map.

convolution kernels with a stride of 4 to the input image. The second layer scales the depth up even further by applying 128 $5 \times 5 \times 48$ convolution kernels [25].

2.3.3 Max-Pooling Layer

Pooling layers are commonly inserted between successive convolutional layers in order to progressively reduce the spatial extent of the data [26]. The common form max-pooling replaces a certain location of the input with the maximum intensity of the area [8]. The operation is extended to higher dimensions by applying independent two-dimensional max-pooling to each depth channel [26]. Fig. 2.19 shows this for a depth-three image.

The output of the pooling layer is controlled by the size of the kernel and the stride between kernel applications. The most common choice is to apply a 2×2 kernel with a stride of 2. With these choices, four pixels in the input are replaced with a single, reducing the amount of data by 75% [26].

Pooling is beneficial not only for increasing computational speed, it also makes the network approximately invariant to smaller translations in its input. This means that even if the input is translated a small distance, most of the outputs of the pooling layer remain the

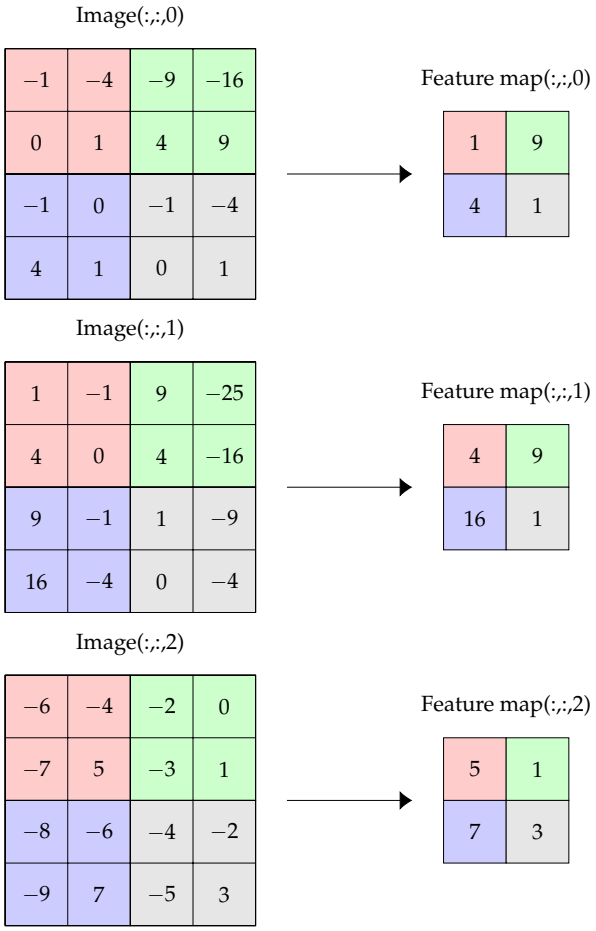


Figure 2.19: Max-pooling of an image with 3 depth channels using a 2×2 kernel applied with a stride of 2. The max operation is applied independently to each depth channel and the result stored in the corresponding depth channel in the output. The colored cells in the output correspond to the 2×2 area with the same color in the image.

same [8]. The approximate translation invariance is illustrated in Fig. 2.20. Pooling also allows for increasing the receptive field of the network, meaning that the information under the kernel is derived from a larger area. While the receptive field can be increased also by making the network deeper, the per-layer increase obtained by applying pooling is larger than that of adding additional convolutional layers [35]. Additionally, the downsampling itself is directly beneficial as it means that the inputs of subsequent layers are smaller, reducing the number of weights required by the network [36].

2.3.4 Fully Connected Layer

Fully connected layers are most commonly used as the last layer of convolutional neural networks [17], although they may less frequently appear earlier as well [36]. In a fully connected layer, each neuron is connected to all neurons in the prior layer [30]. This means that fully connected layers in convolutional networks, much like the layers in feedforward networks, perform a per-neuron scaling. It is common that the large number of parameters required for this scaling is the main contributor to the steep memory requirements of convolutional networks [36]. Examples of this include the AlexNet, which consists of five convolutional layers and three fully connected layers [25]. In this network, 58 000 000 of a total of 60 000 000 parameters correspond to fully connected layers [36]. In the VGG16 network, consisting of

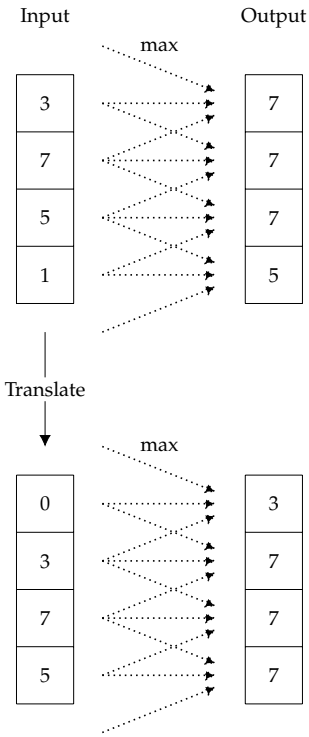


Figure 2.20: Illustration of the approximate translation invariance of max-pooling. The upper image shows the result of applying max-pooling with a 3×1 kernel. In the lower image, the input has been shifted down one step. Despite the change in the input, large parts of the output remain the same.

thirteen convolutional layers and three fully connected ones [24], 123 000 000 of the 138 000 000 total parameters are found in fully connected layers [36].

Convolutional networks are constructed predominantly such that the convolutional or pooling layer directly preceding a fully connected layer in the network outputs an order-one tensor. This is sometimes referred to as flattening [18] and allows for evaluating the fully connected layer as a matrix-vector operation [37]. As an example, the activations $h_j^{(n)} \in \mathbb{R}$, $j = 1, 2, 3$ of the fully connected layer $L^{(n)}$ shown in Fig. 2.21 may be evaluated as

$$\begin{bmatrix} h_1^{(n)} \\ h_2^{(n)} \\ h_3^{(n)} \end{bmatrix} = \Phi \left(\begin{bmatrix} w_{1,1}^{(n)} & w_{1,2}^{(n)} & w_{1,3}^{(n)} \\ w_{2,1}^{(n)} & w_{2,2}^{(n)} & w_{2,3}^{(n)} \\ w_{3,1}^{(n)} & w_{3,2}^{(n)} & w_{3,3}^{(n)} \end{bmatrix} \begin{bmatrix} h_1^{(n-1)} \\ h_2^{(n-1)} \\ h_3^{(n-1)} \end{bmatrix} + \begin{bmatrix} w_{0,1}^{(n)} \\ w_{0,2}^{(n)} \\ w_{0,3}^{(n)} \end{bmatrix} \right) \tag{2.24}$$

where $\Phi : \mathbb{R} \to \mathbb{R}$ is an arbitrary activation function. Here, $h_i^{(n-1)} \in \mathbb{R}$, $i = 1, 2, 3$ is the activation of the i^{th} neuron in layer $L^{(n-1)}$ in Fig. 2.21. The elements of the 3×3 matrix are the layer weights $w_{i,j}^{(n)} \in \mathbb{R}$ and rightmost vector consists of the layer biases $w_{0,i}^{(n)} \in \mathbb{R}$.

2.4 Data-Level Parallelism

Data-level parallelism is a parallel programming model that relies on vectorization techniques to increase program throughput [38]. Rather than scheduling instructions over multiple logical threads of control, it relies on multiple compute units performing the same operation on different data items. The approach is used in GPUs [39] where billions of pixels are processed independently of each other [40].

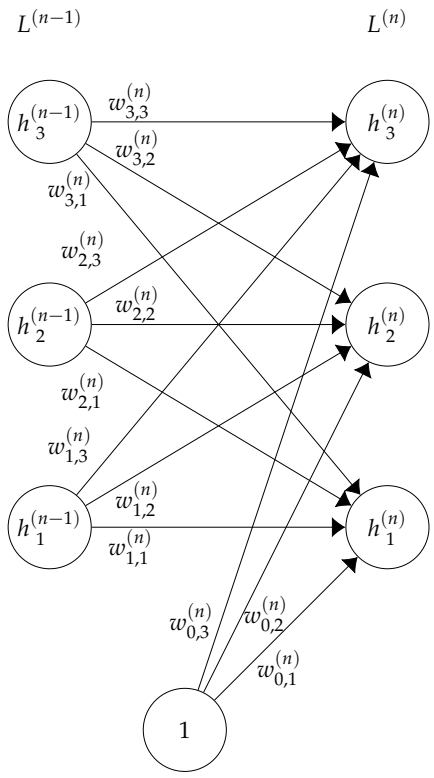


Figure 2.21: A fully connected layer $L^{(n)}$ and its input layer $L^{(n-1)}$, both with their respective neurons arranged in columns.

When compared to concurrency, data-level parallelism has both advantages and shortcomings. Concurrent programming relies on scheduling a potentially large number of logical threads over a fixed number of physical cores. Which threads are run on which core is decided by the operating system scheduler [41]. This task scheduling is often both difficult and computationally expensive [42]. Additionally, sharing data between multiple threads introduces the risk of encountering race conditions [43]. As data parallelism does not rely on threads, it does not suffer from these drawbacks. Its main disadvantages are instead that it generally requires well-structured data access and has limited support for branching [44].

2.4.1 SIMD

SIMD is the perhaps most common form of data-level parallelism. The name is an abbreviation of single instruction stream, multiple data stream [45]. SIMD refers to a subset of data parallelism that allows for executing a single instruction in parallel across multiple data streams [46]. Using the same taxonomy, concurrency-based parallelism is referred to as MIMD, or multiple instruction stream, multiple data stream [45]. Fig. 2.22 illustrates the difference between MIMD and SIMD when adding two vectors **a** and **b** in parallel. The MIMD machine uses a separate thread for each addition whereas the SIMD machine performs the addition in a single thread by relying on vector instructions inherent to the processor.

2.4.1.1 x86-64 Implementation

The SIMD registers of x86-64 processors are divided into lanes, each of which can be used for both scalar arithmetic, Boolean operations, comparisons and data conversions. The number of lanes available depends not only on the processor architecture but also on the size of the

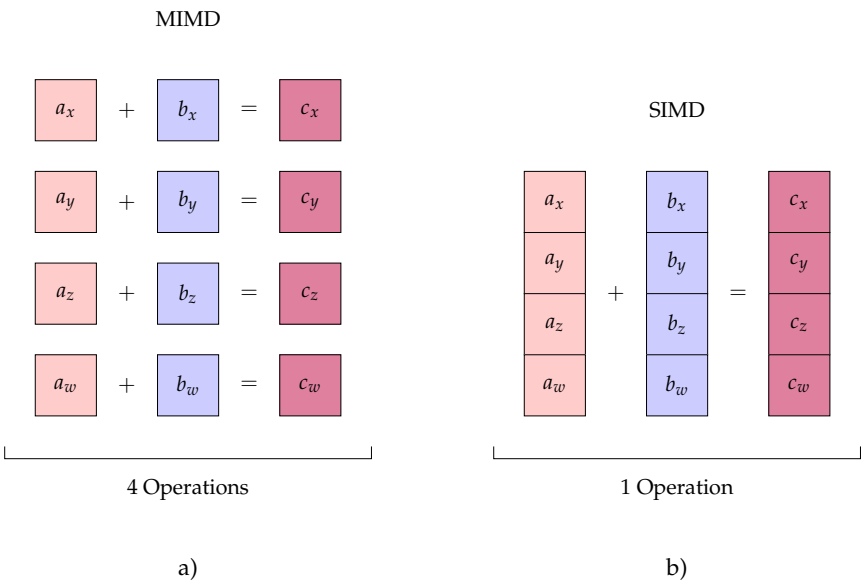


Figure 2.22: Parallel addition of two vectors, each consisting of four elements. a) A MIMD-based approach where each addition is performed by a separate thread. b) A SIMD-based approach, all additions performed in a single instruction by a single thread.

Table 2.1: Fundamental x86-64 data types with C counterparts.

Data Type	**Description**	**C Counterpart(s)**
byte	8-bit integral data type, signed or unsigned	signed char, unsigned char
word	16-bit integral data type, signed or unsigned	short, unsigned short
dword	doubleword, 32-bit integral data type, signed or unsigned	int, unsigned int
qword	quadword, 64-bit integral data type, signed or unsigned	long long, unsigned long long
real4	Single precision (32-bit) floating point data type	float
real8	Double precision (64-bit) floating point data type	double

data type used. As an example, the 128 bit wide xmm$n$, $n \in \mathbb{Z}_0^+, n \leqslant 15$, registers used in the streaming SIMD extensions (SSE) of x86-64 CPUs can at any time be partitioned in one of several ways. They are able to process 16 bytes, 8 words, 4 dwords, 2 qwords, 4 real4s or 2 real8s in parallel [47]. These data types are described in Table 2.1.

SIMD operations are performed on packed data types [47]. Fig. 2.23 illustrates this where the x86-64 instruction pmaxud (packed max of unsigned dwords) is used on the packed registers xmm0 and xmm1. After the instruction has executed, the destination register xmm0 contains the lane-wise max values of the two input registers.

One of the main advantages of SIMD, regardless of processor architecture, is that it potentially allows for reducing the total number of loads from memory. This is done by issuing

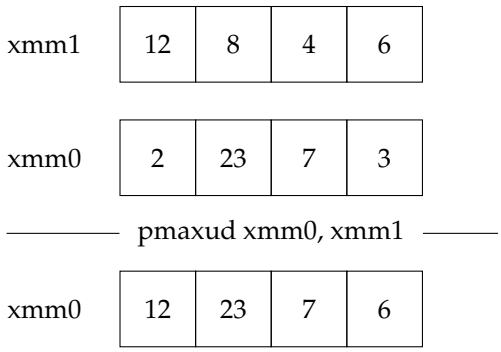


Figure 2.23: Computing the maximum of 4 packed unsigned 32-bit integers using the SSE instruction pmaxud. The two topmost grids show the contents of each of the 4 dword lanes of the registers xmm0 and xmm1 before the max operation. The bottommost grid shows the contents of the destination register xmm0 after the operation.

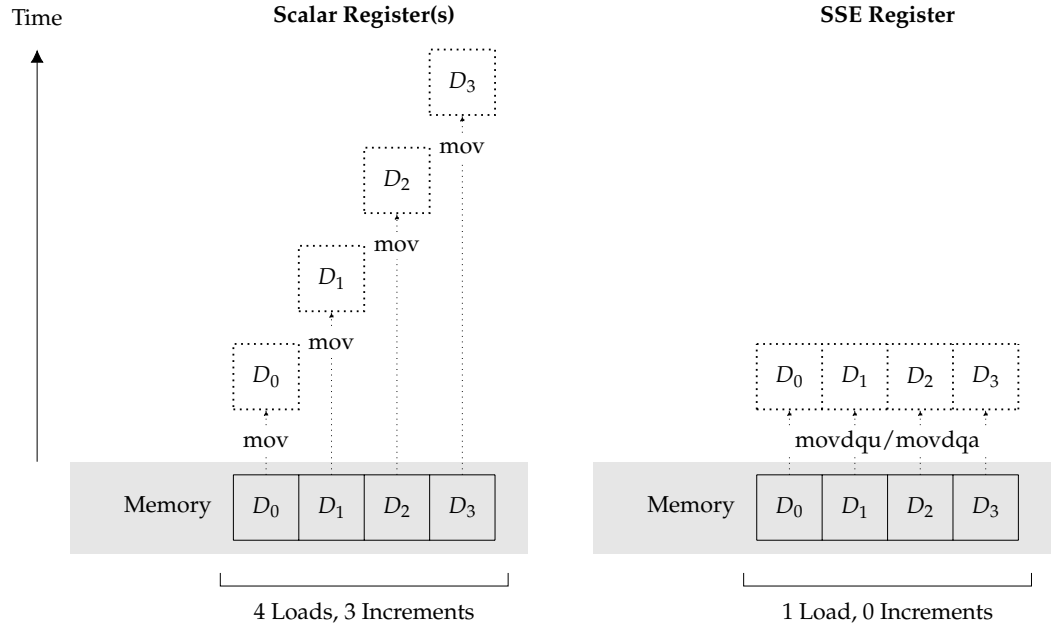


Figure 2.24: Simplified memory access patterns for scalar and vector registers when loading four dwords from memory. Using scalar registers, only one value is loaded per instruction and the memory address must be incremented between each load. SSE registers may instead load four dwords in a single instruction.

instructions that fetch larger chunks of data at once [46]. Fig. 2.24 aims to illustrate this by showing the process of loading four consecutive dwords from memory. When using scalar registers, only a single dword is loaded per instruction. Once a dword has been loaded, the address is incremented by 4 bytes and the next dword loaded. Using packed data transfer with an SSE register, four dwords are loaded using a single instruction. If more data is to be loaded, the address is incremented by 16 bytes, preparing for the next load. This means that using packed data transfer, in this case, reduces the number of memory accesses by 75% and the number of increments by 67%.

Whereas Fig. 2.24 shows the potential gain of using packed data transfer, it should be noted that it illustrates an ideal case. Packed data transfer generally requires that the data

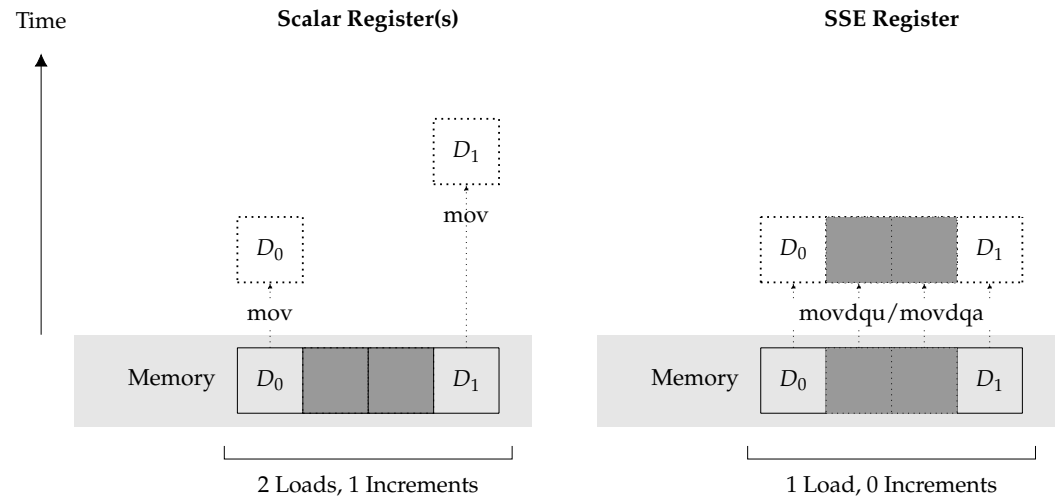


Figure 2.25: Memory access patterns for scalar and SSE registers where every third dword in memory is desired. Unused addresses and register lanes are shown in darker gray. The scalar registers operate by loading one dword at a time and then moving to the next. The SSE register loads 4 consecutive dwords at once but as the desired data elements are farther apart, two dwords are loaded in vain.

be arranged consecutively in memory. If this is not the case, loading larger chunks at once is less beneficial. This is illustrated in Fig. 2.25. Packed instructions are always performed on all lanes of a vector register. This means that even if a register is partially loaded with unwanted data in this manner, the unwanted data is still used by any instructions issued [48]. In practice, this is typically solved by discarding unwanted data by shuffling, packing or extracting register lanes [49].

2.4.1.2 Availability in High-Level Languages

SIMD instructions are available in modern x86 and x86-64 CPUs via the SSE and the more modern advanced vector intrinsics [46]. Despite this, leveraging them in full in high-level languages is typically difficult. While both GCC and LLVM perform limited automatic vectorization when compiling C and C++ [50], such optimizations can only be performed under certain conditions. The complexity in ascertaining that these conditions are met frequently prohibits automatic vectorization altogether [51]. Explicit vectorization is available only by using APIs such as OpenMP and Boost.SIMD [50] or by relying on vendor [52] or compiler [53] intrinsics at the cost of portability. As an alternative to C and C++, Rust offers standardized, albeit both limited and platform dependent, support for vector instructions available on x86 and x86-64 processors [54].

In addition to the selection of SIMD options available in high-level languages being sparse, the ones that are available seldom provide a higher abstraction level than writing assembly would. Furthermore, the concerns regarding portability and forward compatibility that accompany them [50] make leveraging SIMD difficult.

2.5 Computer Memory

Computer memory is often thought of as a flat array of bytes, each of which identified by its address offset from 0 [55]. This model of single-level memory is sufficient early during development but refining it allows for additional increases in program throughput [56].

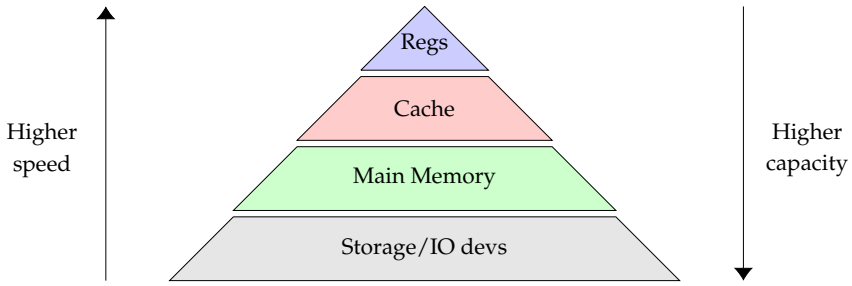


Figure 2.26: A typical memory hierarchy in a modern computer.

The memory hierarchy of a modern computer includes several memory tiers of varying sizes and speeds. Fig. 2.26 shows a typical memory hierarchy where memory speed increases upward and storage capacity downward. The main differences between the levels in the hierarchy are the access speeds, costs per unit of storage and memory volatility [56].

Volatile memory is memory that requires power to retain its information. If the power is lost, so is the content of the volatile memory. Examples of volatile memory include caches and main memory. Non-volatile memory retains its stored information even without power but is typically slower than volatile memory. Common forms of non-volatile memory include mechanical hard drives, solid state drives and optical disks [56].

This section focuses solely on volatile memory, specifically main memory, caches, scratchpads and registers.

2.5.1 Main Memory

Main memory constitutes the primary storage of a computer. It consists of dynamic random access memory (DRAM), commonly referred to as system RAM. While the cost per unit of storage is higher than that of non-volatile storage devices, it is still moderately affordable. This relatively low cost allows for equipping computers with several gigabytes of main memory [56].

The speed of the system RAM is largely determined by the width and speed of the memory bus. The bus width is the number of bits that can be sent to the CPU at the same time. The speed of the memory bus is determined by the number of times a group of bits can be sent to the CPU each second. In practice, the memory bus is rarely the sole factor that determines speed of memory access [56]. Instead, memory latency, the time between a memory request is made and the point when the data transfer is completed [57], plays a significant part. This typically results in data transfer between main memory and CPU becoming a bottleneck [56].

2.5.2 Caches

In order to lessen the impact of memory access latency, caches were introduced. Caches are smaller units of memory consisting of static random access memory (SRAM) located closer to or on the CPU itself. They typically come in several levels which are ordered by access times. SRAM is more expensive than DRAM and as such, caches tend to be significantly smaller than main memory [56].

On the vast majority of modern processors, the different cache levels are all housed on-chip. Level 1 (L1) cache is the fastest but also the smallest. Level 2 (L2) cache is larger and slightly slower [56]. Many modern CPUs provide a third and fourth level of cache that adhere to the pattern of being larger but slower [58]. Whereas L1 and L2 cache is typically core-private, L3 and L4 caches are, when available, usually shared between the cores [56]. Table 2.2 shows the total sizes of the cache levels of an AMD Ryzen 7 3700x processor [59]. It should

Table 2.2: Total cache sizes of an AMD Ryzen 7 3700x processor.

Cache Level	Total Size
L1	512 kB
L2	4 MB
L3	32 MB

be noted that as the L1 and L2 caches are core private and the processor has 8 cores [59], the amount of L1 and L2 cache available to a single core is 64 kB and 500 kB, respectively.

The cache is the first memory that is searched by the CPU when it needs data or instructions. If the required data is in cache, there is no need to fetch it from memory. Accessing data that is already in cache is referred to as a cache hit whereas loading data that is not in cache is called a cache miss [60]. By designing algorithms that minimize the number of cache misses, performance can be increased significantly [55]. In the event of a cache miss, the required data has to be loaded from main memory. Rather than loading just the requested data, a larger block, commonly called a cache line, is transferred into the cache. Assuming subsequent data requests attempt to access data located in the physical vicinity of the previously requested item, this increases the likelihood of cache hits [60].

The content of the cache is typically managed automatically by the CPU using specific replacement policies [61]. The effectiveness of these replacement policies can be increased by the programmer by carefully managing memory accesses. Examples of this includes optimization techniques such as loop blocking, a technique that changes the access pattern of nested loops [62].

2.5.3 Scratchpad Memory

Scratchpad memory is, much like caches, a small, fast on-chip memory. Whereas caches are managed automatically by the processor at runtime, scratchpads are controlled either by the programmer or by the compiler. This makes scratchpads a good fit for applications such as video or digital signal processing [63] where the programmer might be able to manage the memory more efficiently than the processor would.

Scratchpads are common in more specialized processors such as DSPs [64] and GPUs [65] where the potential increase in throughput [63] may outweigh the added workload posed to the programmer. They are also a common choice in embedded devices as they typically consume less power than caches do [63].

2.5.4 Registers

At the very top of the memory hierarchy are the processor registers. These are memory cells built into the CPU, closely connected to the arithmetic and logic unit of the processor. Registers are both very small and very fast and are typically managed by the compiler in higher-level languages. Managing them manually is possible using assembly language or inline assembly in C [56]. While C provides the *register* keyword, this allows for only very limited, if any, actual control over registers, so much so that the keyword has been deprecated in C++ since 2011. It remains valid in C as it has other, potentially useful, effects [66].

Table 2.3 shows the approximate access times to the different levels of the memory hierarchy used in an Intel Xeon E5 v3 processor [67]. Here, LLC is used as an abbreviation for last-level cache. A cycle refers to the smallest period of time that exists to the CPU. The speed of a processor is usually given as the number of such clock cycles the CPU executes each second, commonly called the clock frequency [68]. A processor with a clock frequency of 3 GHz executes 3 billion clock cycles each second. This means that a Xeon processor clocked to 3

Table 2.3: Approximate access times to parts of the memory hierarchy of an Intel Xeon E5 v3.

Memory Type	Access Time
L1 cache	4 cycles
L2 cache	11 cycles
LLC	34 cycles
DRAM	60 ns

GHz would access L1 cache in roughly 1.3 nanoseconds, approximately 45 times faster than a load from main memory.

2.6 Digital Signal Processors

A digital signal processor is a microprocessor optimized for performing operations common to real-time signal processing [69]. These operations may, depending on the design of the processor in question, include anything from voice and data compression and signal multiplexing [3] to multimedia acceleration [70].

DSPs are typically embedded as part of larger systems [69] such as smart TVs or mobile phones where they supplement the primary processor. The reason for adding coprocessors rather than relying only on the CPU is that specialized chips typically outperform the primary processor in aspects such as computational performance [71]. In the case of DSPs their main advantage is that they provide power efficiency superior to that of a general CPU [70]. While attempts to involve DSPs when evaluating deep learning algorithms have been made [72], the prospect remains largely unexplored.

The instruction set of a DSP typically includes instructions that cannot be expressed using common high-level programming languages [5]. In order to fully leverage the capabilities of a DSP, vendor-specific languages or language extensions are used. This usually equates to either writing assembly or using an API that exposes the instructions of the DSP to a higher-level language. Both of these options require knowledge of the architecture of the particular DSP and how to write code that makes best use of its available instructions and hardware units [5].

DSPs come in two different forms, real DSPs and complex DSPs. These are named after the respective types of mathematics they default to — a real DSP works with real number mathematics whereas complex DSPs work with complex number mathematics. The need for complex DSPs exists in part due to it being common for high-speed telecommunications standards to use complex representations of signals [73].

2.6.1 The Harvard Architecture

Many of the tasks for which DSPs are employed require specific mathematical operations to be performed on a continuous stream of data. In cases such as telecommunications and LTE baseband processing, this must be done in real-time [5], lest parts of the continuous input would be lost.

Most conventional desktop processors adhere to the von Neumann architecture where program instructions and data are stored in memory and transferred to the CPU over single instruction and data buses, respectively [74]. DSPs, on the other hand, typically need access to multiple sources of data in each clock cycle. These data might, for example, be a digital signal and the coefficients of a FIR filter that is to be applied to said signal. Ideally, a core should be able to read an instruction and a data element from both the signal and the filter during any given, single clock cycle. As such, DSPs require at least three separate data buses. This architecture is commonly referred to as the Harvard architecture [5].

Table 2.4: Memory sizes of a Texas Instruments TMS320C6713 DSP.

Memory Level	Total Size
L1	8 kB
L2	256 kB

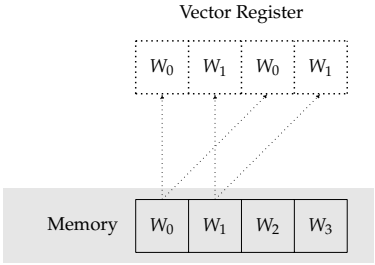


Figure 2.27: Packed data load with a modulo factor 2.

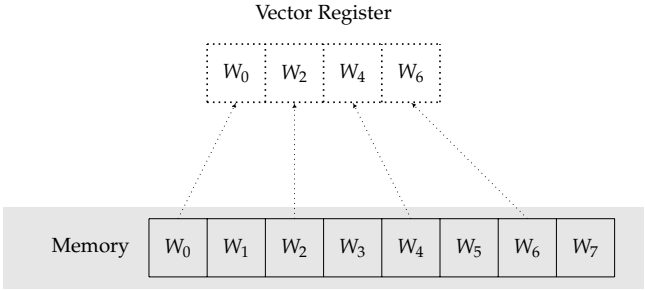


Figure 2.28: Packed data load with memory stride 2.

DSPs employ memory hierarchies consisting of both on- and off-chip memory. The on-chip memory is referred to as L1 memory and is generally very fast. Typically, data can be read from or written to L1 memory in a single cycle. DSPs may optionally employ a slightly slower L2 memory and usually have a significantly larger but slower off-chip L3 memory [5]. Table 2.4 shows the sizes of the levels in the memory hierarchy used in a Texas Instruments TMS320C6713 DSP [75]. It should be noted that, despite being arranged in levels, the memories of a DSP function as scratchpads rather than caches. While DSPs do have caches, these are rarely used for anything but instructions. This stems from the fact that most DSP applications have no need of reusing data. Instead, the data is read from memory and the result computed and written back memory only once [5].

In order to optimize the processing, DSPs commonly need to read and write data in a specific order [5]. An example would be the application of a finite filter to a signal. At some point, all filter coefficients will have been read and the processor should start reading at the first coefficient again. DSPs solve this issue by having special data address generators (DAGs) that support circular addressing. This allows for having the memory address automatically wrap around and start at the top once the data has been depleted [5], much like applying a mathematical modulo operation to the address. Fig. 2.27 shows the result of loading a vector register with a modulo factor of 2.

The DAGs may also be used to read and write data with a programmable memory stride [5]. This may be used to solve the issue of non-consecutive data when performing packed memory loads illustrated in Fig. 2.25 in Section 2.4. An example of this is shown in Fig. 2.28 where the DAG is programmed to load every other word from memory into consecutive lanes of the vector register.

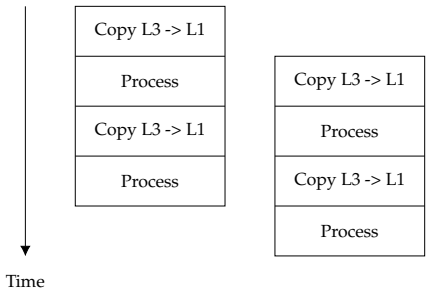


Figure 2.29: Conceptual illustration of asynchronous memory transfer from L3 to L1.

Data transfer is handled by programmable hardware units known as direct memory access (DMA) controllers. These may be preprogrammed to move data in a certain way and to a certain location meaning the DSP does not have to waste cycles performing every data transfer. This allows for the DSP to perform asynchronous buffering where one block of data is processed while the next block is moved into L1 memory by the DMA controller [5]. This is illustrated in Fig. 2.29.

A majority of the workload of DSPs is spent in small loops called kernels. Whereas conventional processors manage loop iterations by decrementing and testing counters, many DSPs have tools that facilitate zero-overhead looping. This means that the looping is handled directly by the hardware, resulting in no cycles being wasted managing loop variables [5].

2.6.2 Memory Mapping

In modern computers, the operating system commonly assigns each running process a protected memory address space. This means that each process has its own stack, heap and data. Any access outside the designated address space of a process is disallowed and typically results in a segmentation fault [55]. The protected memory space effectively acts as a safeguard against a process, knowingly or otherwise, reading and writing the data of another. As might be expected, this comes at a certain cost. In DSPs, where processing efficiency and predictable latency are important, this overhead is rarely tolerated. Instead, resources such as memory, registers and DMA controllers are typically all shared among running threads [5].

2.6.3 Pipelining

Pipelining is a parallelization method where subprocesses are executed in parallel by different hardware units [76]. The approach is used in DSPs in order to increase the raw computational power [77].

Pipelining leverages the fact that successively issued independent instructions may be executed in parallel [76]. Fig. 2.30 shows an example of this where a total $m + 1$ values are processed by the load-store, multiplier and add units. Since these hardware units operate independently of each other, the execution of the separate instructions is overlapped. In the first step, the value x_0 is loaded from memory. The second step consists of a multiplication involving x_0. While this multiplication is performed, the load-store unit loads the next value x_1. The third step involves an addition using x_0. While this addition is performed, the multiplier unit performs a multiplication involving x_1 and the load-store unit loads x_2 from memory. This execution pattern is repeated until the entire input has been processed. The pipeline shown in Fig. 2.30 assumes that each step modifies the value of x_j, $j = 0, \ldots, m$ such that the instructions performed by the add and multiplier units cannot be evaluated concurrently.

While pipelining is often used to increase the throughput of software, there are a number of so-called hazards that may impede performance. These generally manifest themselves when parts of the pipeline do not behave as expected. As an example, a so-called data conflict

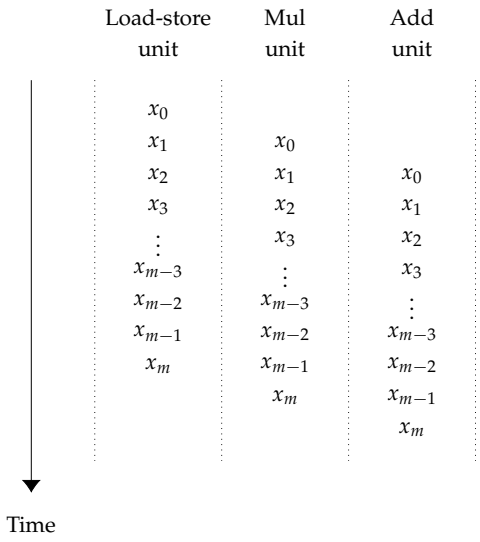


Figure 2.30: Pipelining of the processing the values x_j, $j = 0, \ldots, m$, the instructions issued for each requiring the load-store, multiplier and add units.

```
s ← 0
for i ← 0 to n do
|  s ← s + data[n]
end
```

a)

```
s ← 0
for i ← 0 to n/2 do
|  s ← s + data[2*n]
|  s ← s + data[2*n + 1]
end
```

b)

Figure 2.31: An example of loop unrolling. a) Computing the sum of n elements using a regular loop. b) Computing the sum of n elements using a loop that has been manually unrolled by a factor 2.

occurs when an element produced by an earlier stage of the pipeline is not ready in time. When this issue is encountered, subsequent stages of the pipeline are forced to stall. As a result, computational resources are wasted waiting for the element in question to become ready [77].

2.6.4 Unfolding

Unfolding is a transformation technique commonly used for improving the throughput of DSP software. This is achieved by increasing the number of total tasks in the program [78]. When used on other types of processors, unfolding is commonly referred to as loop unrolling.

The unrolling of a loop results in multiple copies of its body being merged [79]. Fig. 2.31 shows how loop unrolling with a factor 2 may be used when computing the sum of n elements. Here, n is assumed to be even for the sake of simplicity.

Unrolling loops has several potential advantages, including reducing loop variable overhead, avoiding excess copies and improving the efficiency of pipelined functional units [80]. Additionally, unrolling loops may allow for exploiting parallelism that would otherwise be hidden behind loop-carried dependencies [79]. Despite these potential gains, loop unrolling should be used with care as it also increases the size of the code. If the code becomes large enough that the instructions no longer fit in the instruction cache, the performance starts to degrade again [81].

3 Method

This chapter describes the approach used when implementing the machine learning primitives on the DSP. The correctness of the algorithms developed is proven in Appendix A.

Unless otherwise stated, any operation referred to in this chapter is assumed to be packed. As an example, if an instruction is referred to as an addition, this would be a packed, lane-wise addition of two vector registers.

3.1 Memory Management

Memory management is a key aspect when implementing neural networks. This is especially true when working with implementations for a DSP.

The DSP designed by MediaTek employs a multi-tier memory hierarchy comprising both on- and off-chip memory. The on-chip memory functions as a scratchpad. Any data that cannot be held in the off-chip memory must be loaded from disk. Memory reads and writes can only be performed from or to on-chip memory. Transfers between on- and off-chip memory require explicit instructions to be issued.

Since evaluating any non-trivially sized layer of a convolutional network requires more data than can fit in the on-chip memory at a time, the memory has to be managed incrementally. The approach proposed in the thesis assumes that the data required to evaluate a particular layer fits in the off-chip memory.

3.1.1 Single-Use Data

When evaluating convolutional and max-pooling layers, the main contributor in terms of memory usage is the input image. As both convolution and max-pooling work on a per-row basis, the input for these is stored in row-major order as illustrated in Fig. 3.1.

In the case of the fully connected layer, the part with the largest memory footprint is the weight matrix. As the layer is evaluated as a matrix-vector multiplication, a naive implementation would load each element of the input vector from the on-chip memory once for each row of the weight matrix. By rearranging how the weight matrix is stored in memory, the number of times the input vector must be loaded can be reduced significantly. As a result, the weight matrix was not stored in row-major order but rather in an order that facilitates the reuse of input data. This memory layout is explained in detail in Section 3.4.

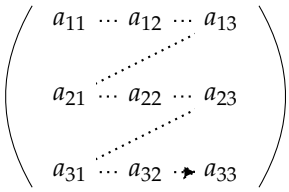


Figure 3.1: Row-major storage of a two-dimensional array.

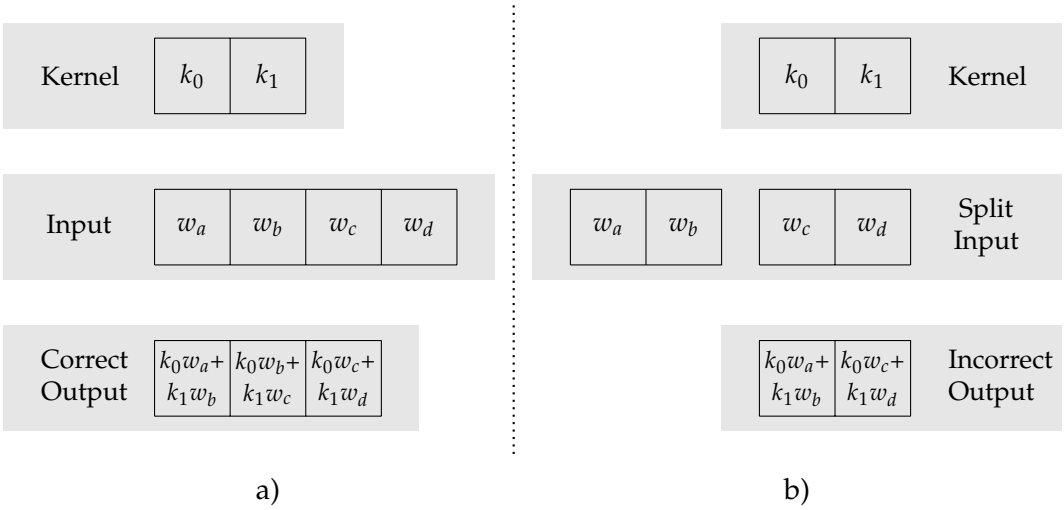


Figure 3.2: One-dimensional cross-correlation. a) The input is intact and the output is correct. b) The input is split in the middle without overlap. As the kernel can only be applied to one of the halves at a time, the pair w_b, w_c is lost and the output is incorrect.

While loading data from on-chip memory is fast, data transfer between on- and off-chip memory is slower. In order to hide the latency of these transfers and reduce the time the processor spends stalling, the asynchronous buffering capabilities of the processor are employed. This is realized by partitioning the space reserved for the input in the on-chip memory in two halves. While the data is copied from the off-chip memory to one of the halves, the other half can be used for computations.

As transferring data from off- to on-chip memory is comparatively expensive another central objective is to transfer each data item in the input only once. This is attained by computing the maximum number of rows that fit in the on-chip memory at one time and transferring that number of rows all at once. The number of data items is rounded down to a multiple of the width of the input. This is done as transferring anything but full rows would require overlapping the data in order to not lose information. Fig. 3.2 illustrates the issues of splitting a row without data overlap.

Convolutional and max-pooling layers work with a receptive field that potentially spans several rows of the input image. This means that in order to transfer every input row only once, it has to be possible to read from both of the on-chip memory halves when processing the bordering rows. Fig. 3.3 shows one such situation.

When processing the bordering rows of the two halves, two properties have to be ensured. Firstly, any transfer from off- to on-chip memory has to be completed before the border processing commences. This is solved by inserting a memory fence that causes the processor to stall until any such transfer has been completed. Secondly, it has to be guaranteed that processing of all the bordering rows has been completed before the next data transfer is initiated.

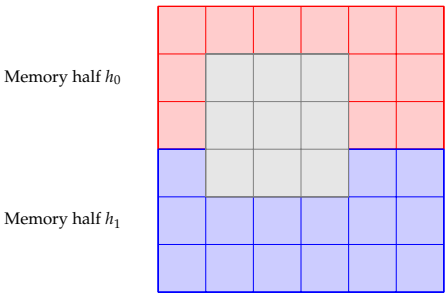


Figure 3.3: A 3×3 filter kernel overlapping the two memory halves.

This is achieved by simply deferring the start of the subsequent memory transfer until the bordering rows have been processed.

3.1.2 Frequently Used Data

Not all data can be used only once and then discarded. Instead, the filter coefficients of the convolutional layers and the input vectors of the fully connected layers have to be reused multiple times. Copying this data from off- to on-chip memory each time it is to be used would be prohibitively expensive. As such, it is transferred to the on-chip memory early and kept there for the duration of the layer evaluation.

The filter coefficients of the convolutional layer may be stored even more efficiently. While loading data from the on-chip memory is fast, any excess time spent performing these loads remains a waste of execution time. In order to avoid this, the coefficients are stored in scalar registers of the processor. This means that, provided that the kernel is small enough to fit in the available registers, no cycles are wasted loading filter coefficients multiple times.

3.1.3 Output Data

The on-chip memory cannot be used exclusively for input data and weights, it also has to hold parts of the computed output. As with the input, it is desirable to avoid processor stalls caused by writing from on- to off-chip memory. In order to achieve this, the same asynchronous buffering used for the input is employed also for the output data. As with the input, the output is written back to the off-chip memory in rows.

3.2 Convolutional Layer

The convolutional layer proved to be the most time-consuming to design. This is mainly due to the large number of varying parameters involved. Certain parameter configurations require solutions that have a significant performance impact. As such, it is desirable to avoid these unless they are absolutely necessary. This is achieved using metaprogramming techniques for developing partial specializations of the algorithm. This section is arranged in accordance with the different specializations developed. While strided and padded convolution are described separately, the methods detailed in the respective sections may be combined to allow for both an arbitrary stride and an arbitrary choice of padding. It should be noted that the convolutional layers are implemented using cross-correlation rather than proper convolution. Furthermore, zero padding is the only form of padding considered.

One of the aspects that remains consistent across the different specializations is the memory management. As mentioned in Section 3.1.2, the filter coefficients are loaded into the scalar registers of the processor early during evaluation. As the input and output can be treated as single-use data, they are managed as described in Section 3.1.1.

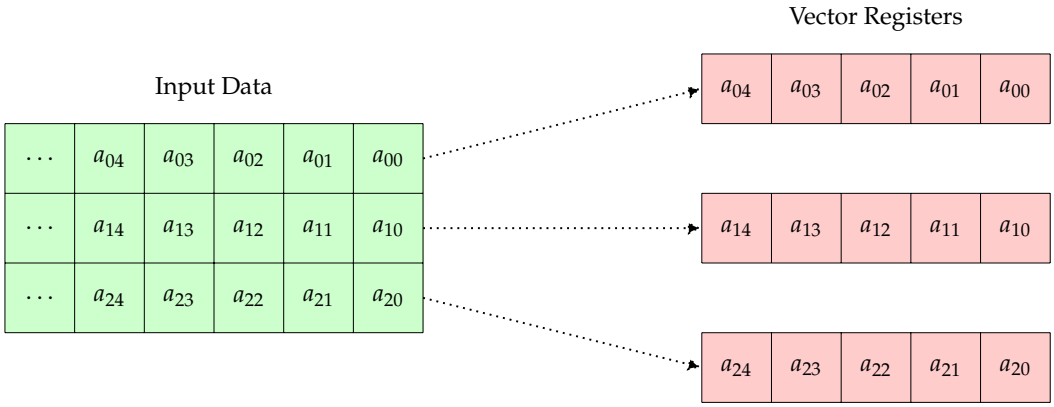


Figure 3.4: Input data loaded into 5-lane vector registers.

3.2.1 Simple Convolutional Layer

The simplest form of convolutional layer implemented was that when the stride was 1 and the padding 0. When convolving the input image with an $m \times n$ filter kernel, simultaneous access to m rows of the input is needed. The processing of each row of the output is performed in a zero-overhead hardware loop. Each iteration, values from each of the input rows are loaded from the on-chip memory. Fig. 3.4 shows how the data is loaded into vector registers. For the sake of lucidity, the registers shown comprise only 5 lanes.

The approach used for evaluating a convolutional layer with stride 1 and padding 0 is shown in Algorithm 1. The upper limit of the inner loop $(NCols + 63)/64$ effectively rounds the fraction $\frac{NCols}{64}$ up rather than down as would usually be the case with integer division. Vector registers are denoted using $v_{...}$ and scalar registers using $s_{...}$. The notation $v \leftarrow [0]$ means zeroing all lanes of vector register v and $v[a:b]$ is used to signify lanes $a, a+1, \ldots, b-1, b$ of v. Hardware loops are shown as **hwfor**. Loops denoted as ***for** are unrolled using metaprogramming techniques, meaning they are not actual loops in the generated code.

Algorithm 1: Evaluation of a convolutional layer with stride 1 and padding 0.

Result: Convolved feature of the input
for $i \leftarrow 0$ **to** $NRows - NKernelRows + 1$ **do**
  for $r \leftarrow 0$ **to** $NKernelRows$ **do**
    let $v_{persist_r} \leftarrow$ LoadFromRow(i+r, input, 16)
  end
  hwfor $j \leftarrow 0$ **to** $(NCols + 63)/64$ **do**
    let $v_{acc} \leftarrow [0]$
    ***for** $r \leftarrow 0$ **to** $NKernelRows$ **do**
      let $v_{in_r}[0:79] \leftarrow$ Cat($v_{persists_r}[0:15]$, LoadFromRow(i + r, input, 64))
      let $v_{persists_r} \leftarrow v_{in_r}[64:79]$
      ***for** $c \leftarrow 0$ **to** $NKernelCols$ **do**
        let $v_{acc} \leftarrow v_{acc} + v_{in_r} s_{coeff_{r,c}}$
        let $v_{in_r} \leftarrow$ RotateRight(v_{in_r})
      end
    end
    Store(v_{acc}, 64)
  end
end

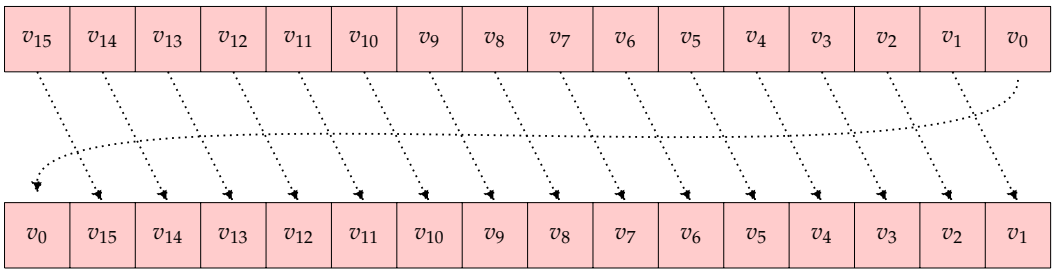


Figure 3.5: Logical right rotation of a vector register.

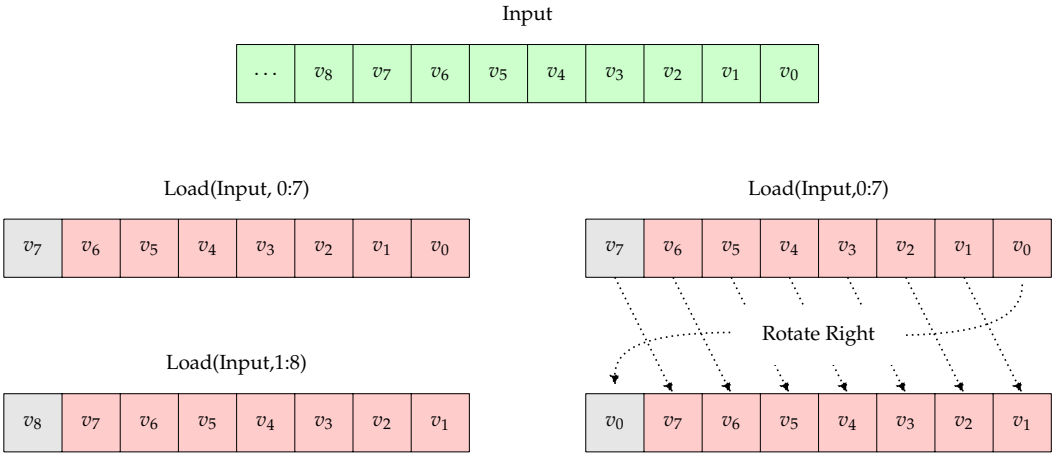


Figure 3.6: Illustration of how packed rotations may be used to reduce the number of memory loads. The two registers on the left are filled by issuing separate loads. The registers on the right are instead populated by a single load and a right rotation. The contents of the red lanes on the respective rows are the same.

The convolution itself is performed in a per-row fashion using logical rotations. Each iteration, all lanes of an input register are multiplied with one of the filter coefficients. The result of the multiplication is stored separately. Once the input has been multiplied with the first filter coefficient, the input vector is rotated one lane to the right. Such a rotation is shown in Fig. 3.5. The post-rotation index i_{n+1} of a register lane is computed from its pre-rotation index i_n using

$$i_{n+1} = (i_n - 1) \ mod \ 16. \tag{3.1}$$

The rotation of the register allows for avoiding additional loads of data. Once 16 lanes have been loaded into a vector register, lanes 1 through 15 contain the data that would be in lanes 0 through 14 for the subsequent load. By rotating the register one lane to the right, the values that were in lanes 1 through 15 are moved to lanes 0 through 14. As such, the rotation can be used to replace the load at the cost of lane 15 containing stale data. Fig. 3.6 illustrates this. Here, each register comprises only 8 lanes due to size constraints.

Once the input register has been rotated, each lane of the register is multiplied with the second filter coefficient for the particular row and the product added to the result. This procedure is repeated for the entire horizontal extent of the filter kernel. Once a row has been processed completely, the result of the next row is evaluated using the same approach. The procedure is repeated until the entire filter kernel has been applied. Fig. 3.7 shows a single application of a 1×3 filter kernel to a one-dimensional signal using this approach. The registers shown contain only 5 lanes for the sake of lucidity.

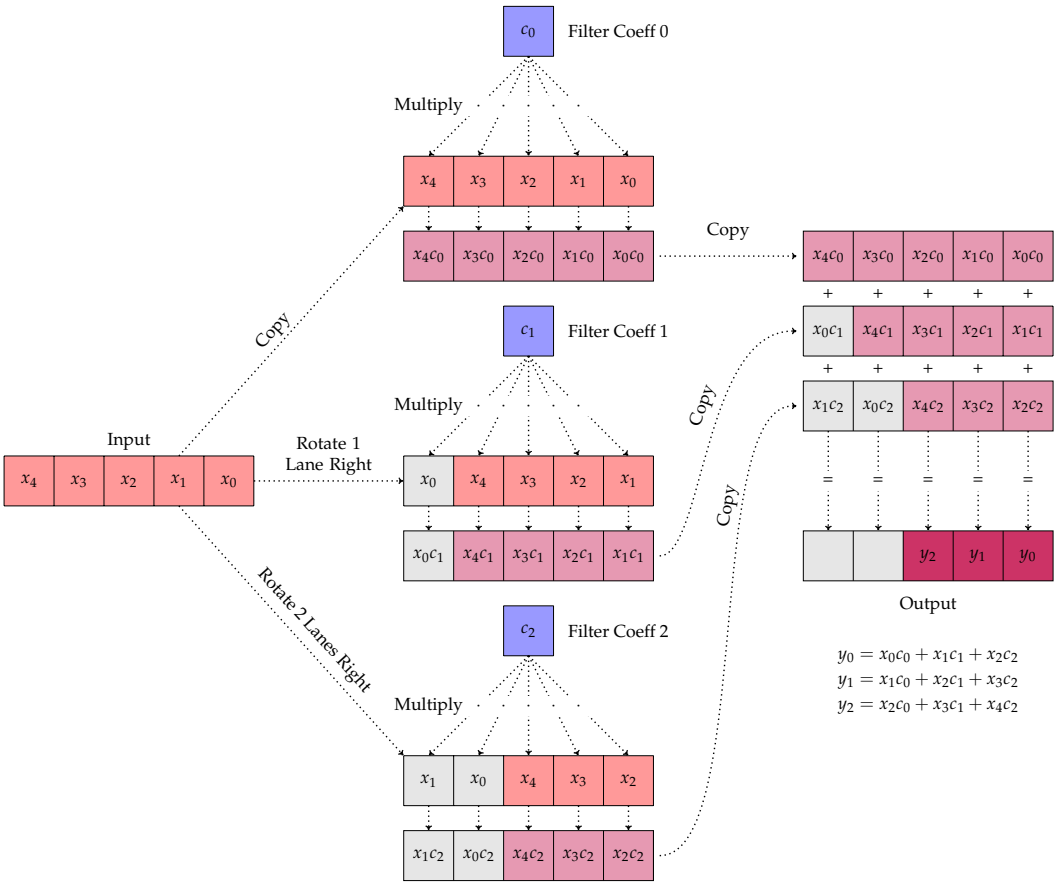


Figure 3.7: Computing the cross-correlation of a one-dimensional signal and a 1×3 filter kernel using packed rotations, multiplications and additions. x_i, $i = 0, 1, \ldots, 4$ is the i^{th} sample of the input signal, c_j, $j = 1, 2, 3$ the j^{th} filter coefficient and y_j the j^{th} component of the output. The content of gray lanes is unused.

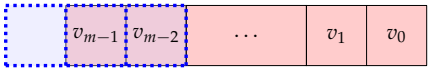


Figure 3.8: Kernel position for the attempted $m - 1^{\text{th}}$ application of a 1×3 kernel to the data contained in an m-lane vector register. The kernel position is shown in blue.

Using vector registers for computing convolution comes with a caveat — filtering data that is loaded partially into several registers cannot be done. As an example, assume one-dimensional convolution with a 1×3 kernel and vector registers comprising $m \in \mathbb{Z}^+, m \geqslant 3$ lanes. Once data has been loaded into the register, the kernel may be applied $m - 3 + 1 = m - 2$ times. For the hypothetical $m - 1^{\text{th}}$ application, the register contains only enough data to multiply with coefficients 0 and 1 of the filter. This is shown in Fig. 3.8. In order to properly convolve the entire signal, the data would have to be loaded with an overlap. This means that every single load from the on-chip memory would have to be preceded with an adjustment of the memory address. Having to manage memory addresses would, in turn, prohibit the use of hardware loops, significantly reducing the throughput of the implementation.

In order to solve the issue with kernel overlap between registers, a larger number of registers are used. Instead of loading values into only a single register, enough data to fill five registers is loaded, all at once. These registers are treated as a single, compound register. Of the five registers loaded, the values of the fifth are copied to a separate register whose content

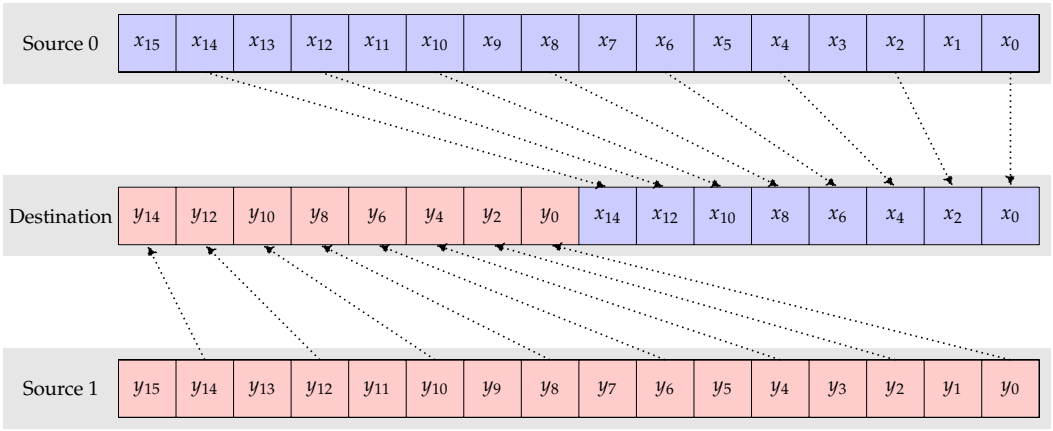


Figure 3.9: Packing of two vector registers.

is kept persistent across iterations of the hardware loop. The loop itself processes 64 filter applications using the algorithm described, requiring data from the entire compound register. For the subsequent iteration of the hardware loop, the values that are stored in the persistent register are moved into the low lanes of the compound register. The remaining lanes are filled with the next values to be read from the on-chip memory. This way, data is loaded continuously from the on-chip memory, requiring neither adjustments to memory addresses nor values to be loaded more than once. The solution works under the assumption that the filter kernel has a width of at most the number of lanes in a single vector register.

3.2.2 Strided Convolution

Supporting an arbitrary choice of stride requires additional work to be performed by the processor. The approach outlined in this section assumes that the stride is at most the spatial width of the filter kernel. This assumption is reasonable as a choice of larger stride would lead to a loss of data. The algorithm still produces the correct result even when the assumption does not hold but it will do so while performing unnecessary computations.

Evaluating strided convolution is done largely as described in Section 3.2.1. The difference is that after the convolution has been computed, the desired values are not stored consecutively in the register. Instead, assuming a stride $n \in \mathbb{Z}^+, n > 1$, the desired values are found in lanes $0, n, 2n, \ldots, n(\lfloor \frac{64}{n} \rfloor - 1), n\lfloor \frac{64}{n} \rfloor$. As such, a total of $\lfloor \frac{64}{n} \rfloor$ values with a period of n have to be written to the result. In order to facilitate the writing of data, the lanes have to be restructured such that the desired values are packed in the $\lfloor \frac{64}{n} \rfloor$ low lanes. Fig. 3.9 shows how the data in two vector registers is to be rearranged if the stride is 2.

As the processor had limited support for rearranging the lanes as required, a new instruction was added to address this need. The instruction in question is described in Section 3.5.1. Once the values have been rearranged as required, the result is written back to the on-chip memory.

It should be noted that the approach used to achieve strided convolution does not perform less work than the non-strided counterpart. Instead, it evaluates the convolution of the layer with a stride 1 and discards unwanted results. Computing the excessive results is a byproduct of evaluating the convolution through vector processing. Nevertheless, it remains a theoretical waste of processing power.

3.2.3 Padded Convolution

Perhaps the most trivial way of evaluating padded convolution would be to actually pad the data with zeroes and then use the approach outlined in Section 3.2.1. Padding the input on

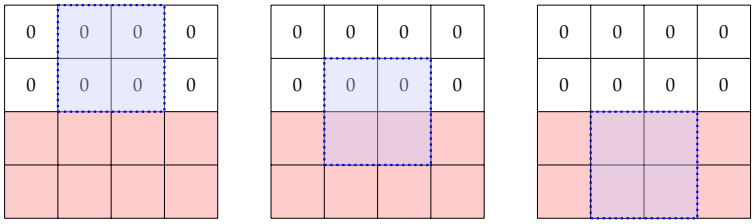


Figure 3.10: The three possible overlap configurations for vertical padding at the top edge of an image. The red cells are part of the input and the cells with zeroes are part of the padding. The position of the kernel is shown in blue.

a DSP would require the additional steps of copying the data from off- to on-chip memory, applying the padding and copying it back to the off-chip memory. It would only be once this was done that the algorithm described in Section 3.2.1 could be used. This approach would violate the DSP philosophy of reading and processing each data item only once. Additionally, having to temporarily write data back to the off-chip memory would incur a significant performance overhead.

Instead of performing the prohibitively expensive actual padding of the input, the padding is emulated. How this is achieved differs depending on how the filter kernel is positioned with respect to the padding.

3.2.3.1 Horizontal Padding

Horizontal padding is the padding that occurs at the left and right edges of the input image. Evaluating a horizontally padded row of the input differs from evaluating a non-padded row only for the first and last iterations of the hardware loop. In order to emulate the zero padding, it must be ensured that the register is loaded partially with zeroes during these iterations. As such, assuming a padding $p \in \mathbb{Z}^+$, the implementation loads p fewer values than can fit in the register during the first and last iterations. The remaining lanes are zeroed.

Ensuring that the vector register contains additional zeroes during the first and last iterations as described allows for emulating the horizontal zero padding of the input. With this, the algorithm described in Section 3.2.1 can be used to evaluate the contribution to the convolved feature.

3.2.3.2 Vertical Padding

Vertical padding refers to padding that occurs at the top and bottom of the image. The filter kernel may at any application be positioned such that it overlaps either entirely, partially or not at all with the image. The different possible configurations for vertical padding at the top of an image are shown in Fig. 3.10. The configurations for padding at the bottom are the inverse of the ones illustrated.

If there is no overlap at all between the filter kernel and the image, the contribution to the convolved feature is always zero. As such, there is no reason to perform any computations for this configuration. Instead, a number of zeroes equal to the width of the convolved feature are written to the result.

If no part of the filter kernel overlaps with the vertical padding, the convolution may be treated as if there were no vertical padding. In this case, the convolution is performed as described in Section 3.2.3.1.

For partial vertical overlap, the contribution of the rows of the kernel that do overlap with the image has to be computed. As the contribution for all values in rows that do not overlap with the image is zero, no computations are performed for these rows. As such, the first step

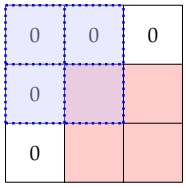


Figure 3.11: A filter kernel positioned such that both vertical and horizontal padding must be emulated. The red cells are part of the input and the cells with zeroes part of the padding. The kernel position is shown in blue.

in evaluating the convolution using a kernel with a partial vertical overlap is to compute the number of rows that do overlap.

Once the number $n \in \mathbb{Z}_0^+$ of overlapping rows has been calculated, the contribution of these is computed. If the part of the image being processed is at the top, then the n bottom rows of the filter kernel are used for the computations. If the part of the image processed is the bottom, the n top rows of the filter are used instead. The evaluation of the partially overlapping rows is performed as described in Section 3.2.3.1.

Using this approach, the case where both vertical and horizontal padding is required is handled automatically. Fig. 3.11 shows such an overlap configuration. Here, the top row contains all zeroes and can be disregarded. The second row of the filter overlaps partially with the image and as such, it is evaluated using the horizontal padding procedure described in Section 3.2.3.1. The evaluation of vertically overlapping rows accounts for horizontal padding. As such, the computed result is correct also when evaluating configurations where the kernel overlaps with both the vertical and horizontal padding.

3.2.4 Higher-Order Convolution

Three-dimensional convolution can be expressed as the sum of the results of several two-dimensional convolutions. As such, the evaluation of each depth channel can be viewed as a separate convolutional layer. Considering this, a convolutional layer of a depth $d \in \mathbb{Z}^+, d > 1$ can be evaluated as d separate convolutional layers, each of depth 1. In other words, no separate implementation has to be developed. Instead, the algorithms described throughout Section 3.2 can be used without modification. Once the d separate depth-1 convolutions have been computed, their respective results are summed to produce the layer output.

3.2.5 Special Considerations

One key aspect that was considered is the overlap configurations when computing strided and padded convolution. A stride larger than one means that the number of vertical overlap configurations is less than the extent of the padding. As the padding is known at compile-time, metaprogramming is used to compute the number of overlapping filter rows without impacting the runtime of the implementation.

The evaluation of a convolutional layer consists of a large number of multiplications, additions and rotations intermixed with memory loads. In order to facilitate this, a separate compound instruction was added. This instruction is described in Section 3.5.2.

3.3 Max-Pooling Layer

The implementation of the max-pooling layer differs depending on the size of the filter kernel and the stride. Throughout this section, it is assumed that the filter kernel is square and that the stride is the same as the width of the kernel. This number is referred to as $n \in \mathbb{Z}^+$.

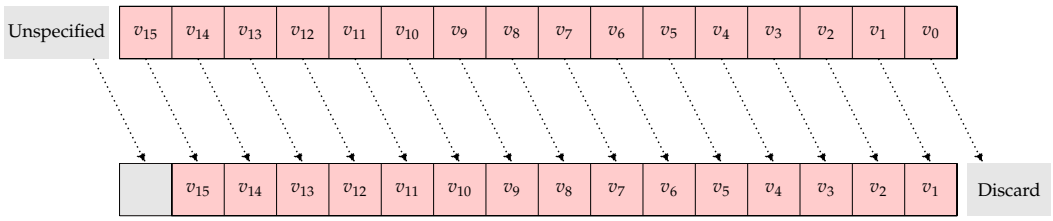


Figure 3.12: Packed logical right shift of a vector register.

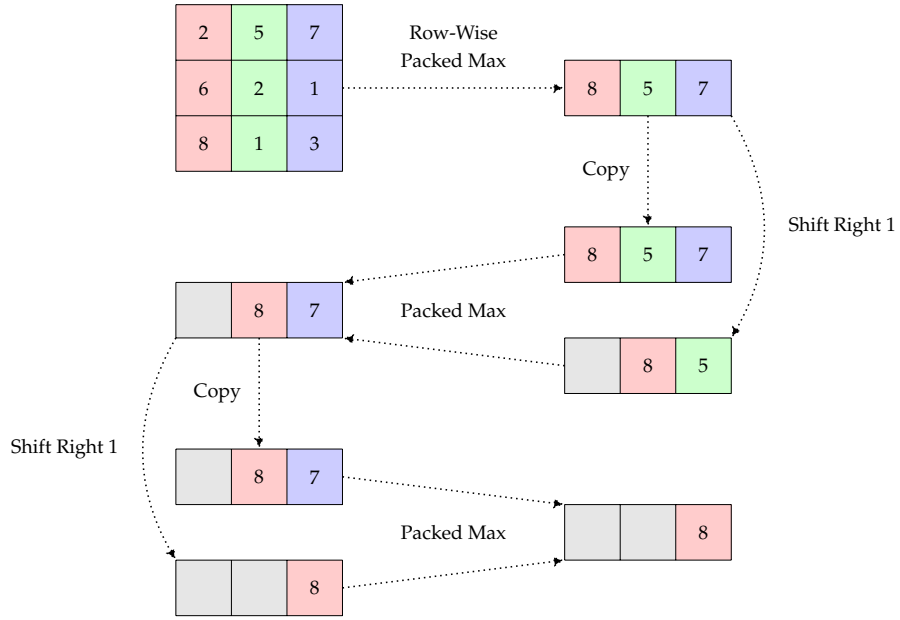


Figure 3.13: Computing the maximum value of a 3×3 grid using logical right shifts and packed max operations. The content of gray lanes is unspecified.

As the pooling operation is performed over an $n \times n$ neighborhood, data from n rows of the input has to be available in the on-chip memory at the same time. This is achieved by using the memory access pattern detailed in Section 3.1.1.

The majority of the workload during the evaluation is spent in a hardware loop. First, data from the first two rows that are to be filtered is loaded from memory. This is followed by a packed, lane-wise max computation. Once the lane-wise maximums of the first two rows have been computed, values from the remaining $n - 2$ rows to be filtered are loaded one row at a time. Between each load, the lane-wise maximums of the result and the newly loaded data are calculated. This allows for computing the column-wise maximums of n rows using only 2 vector registers.

Once the maximums of the rows have been computed, finding the global maximum under the filter kernel is done by computing the maximum over n adjacent lanes in the first register. A simple approach for achieving this is to copy the content of the register to a second register, shifting this second register right and computing the lane-wise maximum of the two. The logical right shift of a register is shown in Fig. 3.12. The procedure of copying and shifting the result register is then repeated a total of $n - 1$ times. Once this is done, the global maximums of $\lfloor \frac{16}{n} \rfloor$ values are held in every n^{th} lane of the original register. The steps of this algorithm are illustrated in Fig. 3.13.

A convenient property of using the approach shown in Fig. 3.13 is that if enough values for several kernel applications fit in a single vector register, the results of these applications

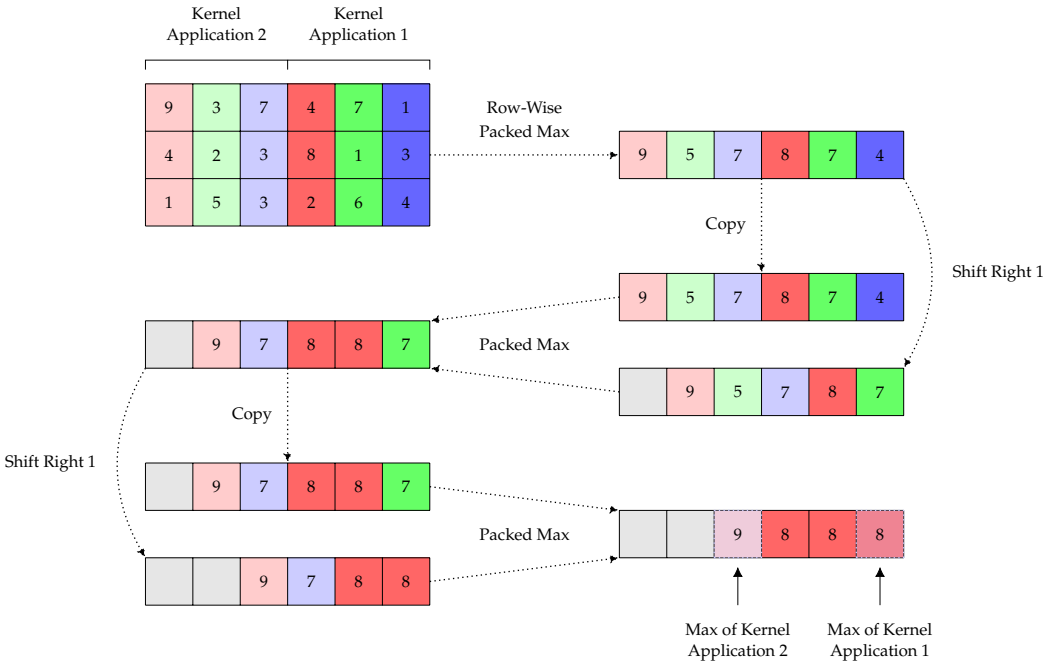


Figure 3.14: A 6-lane vector register used to compute the respective maximums of two 3×3 areas simultaneously.

are all evaluated simultaneously. This is illustrated in Fig. 3.14 where a six-lane vector register is used to compute the respective maximums of two 3×3 areas.

While the algorithm described thus far does produce the correct result, it issues an unnecessarily large number of instructions. Assuming m-lane vector registers, $m \in \mathbb{Z}^+, m > 1$, each packed max instruction computes a total of m maximums. The approach shown in Fig. 3.13 makes use of only one such computation each iteration. This means that in order to compute the maximum over n adjacent lanes, $n - 1 \in O(n)$ packed max instructions have to be issued. This number can be reduced. If the number of adjacent values to compute the maximum of is an even number $2k$, $k \in \mathbb{Z}^+$, the register can be shifted k lanes instead of only 1. If the number of remaining values is instead an odd number $2k + 1$, a single-lane shift has to be used lest some values would be considered incorrectly in subsequent instructions. After this single-lane shift, the number of remaining values to consider is $2k + 1 - 1 = 2k$. Since $2k$ is even, the subsequent shift can again be performed by shifting k lanes.

The improved algorithm requires at most $2\lfloor \log_2(n) \rfloor \in O(\log(n))$ max instructions for computing the lane-wise maximum. Furthermore, it is guaranteed to issue at most the same number of max instructions as the linear version, even when n is small. Fig. 3.15 shows how the improved algorithm is used to compute the maximum of a 7×7 neighborhood.

The principle used to compute the maximum values under the kernel can be summarized as

$$\max\left(\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}\right) = \max\left(\max\left(\begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix}\right), \max\left(\begin{bmatrix} a_{12} \\ \vdots \\ a_{m2} \end{bmatrix}\right), \dots, \max\left(\begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix}\right)\right) \tag{3.2}$$

where the computation of the maximum value of an $m \times n$ matrix is performed in two steps. The first step computes the n column-wise maximums of the matrix. The global maximum of the matrix is obtained by computing the maximum of the column-wise maximums. Adhering

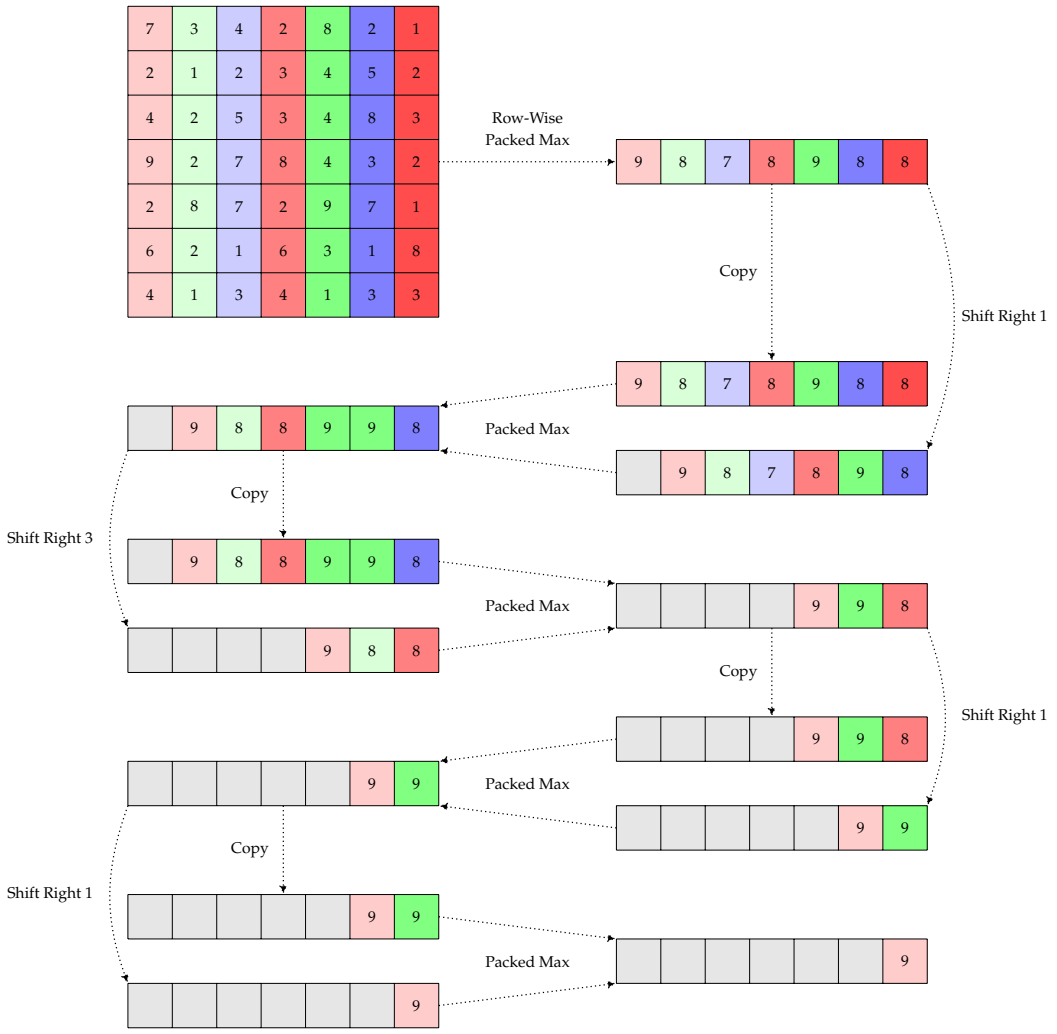


Figure 3.15: Computing the maximum value of a 7×7 grid. Each iteration of the horizontal max computation shifts the largest number of lanes possible in order to minimize the number of issued instructions.

to this approach, it is not guaranteed that all values are compared to each other. In other words, the algorithm is greedy.

Once the maximum values have been computed, they have to be written back to the on-chip memory. At this point, the maximum values are stored in every n lanes of the register. This means that the values have to be rearranged such that the data is stored in the low consecutive lanes of the register. As this issue has already been solved as part of the strided convolution, the solution used is that described in Section 3.2.2 of using the newly added instruction detailed in Section 3.5.1.

Once the values have been moved to the low lanes of the register, all that remains is to write them back to the on-chip memory. The entire algorithm is shown in Algorithm 2. Here, it is assumed that n is 2. For the sake of brevity, management of on- and off-chip memory is not shown. Instead, the function LoadFromRow is assumed to perform everything needed to load data from the row among those to be filtered that is specified by its first parameter. Similarly, any copying of the output from on- to off-chip memory is performed as part of the call to Store.

Algorithm 2: Evaluation of a max-pooling layer with a 2×2 kernel and stride 2.

```
Result: Input downsampled to 25% of original size
for i ← 0 to NRows/2 do
    hwfor j ← 0 to (NCols + 15)/16 do
        let v0 ← LoadFromRow(0, Input, 16)
        let v1 ← LoadFromRow(1, Input, 16)
        let v0 ← Max(v0, v1)
        let v1 ← ShiftRight(v0, 1)
        let v0 ← Max(v0, v1)
        let v0 ← Pack(v0, v1)
        Store(v0, 8)
    end
end
```

3.3.1 Higher-Order Pooling

Computing the output of a max-pooling layer with an input of depth $d \in \mathbb{Z}^+, d > 1$ can be done by evaluating d separate depth-one pooling layers. Since the output of each depth slice is independent of the others, no computational power is wasted using this method. As such, max-pooling with a depth d is implemented by repeatedly applying the approach described without modification.

3.3.2 Special Considerations

During profiling, it was found that handling pooling differently depending on the size of the kernel used allowed for increased throughput. More specifically, the number of active cycles required using kernels of odd spatial extents could be reduced by manually unrolling the hardware loop by a factor 2. This means that instead of each iteration of the loop producing one value of the output, it produces two. This effectively doubles the work performed in each loop iteration. As a consequence, the total number of iterations required is halved.

3.4 Fully Connected Layer

The fully connected layer is implemented as a matrix-vector multiplication followed by a vector addition. This can be summarized as

$$\begin{bmatrix} a_1^{(n)} \\ \vdots \\ a_m^{(n)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{(n)} & \dots & w_{1,k}^{(n)} \\ \vdots & \ddots & \vdots \\ w_{m,1}^{(n)} & \dots & w_{m,k}^{(n)} \end{bmatrix} \begin{bmatrix} h_1^{(n-1)} \\ \vdots \\ h_k^{(n-1)} \end{bmatrix} + \begin{bmatrix} w_{0,1}^{(n)} \\ \vdots \\ w_{0,m}^{(n)} \end{bmatrix} \tag{3.3}$$

where $a_i^{(n)} \in \mathbb{R}$, $i = 1, 2, ..., m$ is the value of neuron i in layer $L^{(n)}$ prior to begin activated. The $m \times k$ matrix is composed by the layer weights $w_{i,j}^{(n)} \in \mathbb{R}$, $i = 1, 2, ..., m$, $j = 1, 2, ..., k$. The j^{th} layer input is denoted as $h_j^{(n-1)} \in \mathbb{R}$ and the rightmost vector consists of the layer biases $w_{0,i}^{(n)} \in \mathbb{R}$.

As described in Section 3.1.2, both the input and bias vectors are kept in the on-chip memory for the entirety of the evaluation to reduce the impact of memory latency. The remainder of the on-chip memory is used for the output and weights.

The majority of the workload of the fully connected layer is spent evaluating the matrix-vector multiplication in (3.3). The multiplication of the input with a row of the matrix corresponds to a component-wise multiplication followed by a summation over the elements in the resulting vector. This type of operation is typically referred to as a MAC operation [82].

A simple implementation of the MAC operation would rely on a hardware loop running over an entire row of the weight matrix. In each iteration, enough values to fill a vector register would be loaded from the input vector and the same number of values from the weight matrix. This would be followed by a multiplication of the loaded input and weight data, the result of which would be added to an accumulator register.

The MAC approach outlined performs the accumulation over multiple steps. Rather than relying on a single accumulator variable, the accumulation is performed partially in each lane of the accumulator register. Once an entire row of the matrix has been processed, the lanes of the accumulator register are summed, producing a scalar. This relies on the mathematical equivalence

$$\sum_i \mathbf{u}_i + \sum_i \mathbf{v}_i = \sum_i (\mathbf{u} + \mathbf{v})_i \tag{3.4}$$

where $(\mathbf{u} + \mathbf{v})_i$ denotes the i^{th} component of the sum of two general vectors $\mathbf{u}$ and $\mathbf{v}$. Here, the left-hand side translates to accumulation using a single variable. In the right-hand side, a partial accumulation is performed through the vector addition. The components of the resulting vector are then summed, yielding the same value as the left-hand side. Despite (3.4) showing a mathematical equivalence, the results may in practice differ between the two sides of the equation due to differences in floating point rounding.

The approach outlined above has one glaring flaw, namely that each element of the input vector has to be loaded from memory once for each row of the weight matrix. The main cause is that the data is stored in simple, row-major order. By rearranging the weight data, the number of times the input values have to be loaded from the on-chip memory can be reduced.

Instead of storing the weight data in row-major order, it is stored in something resembling blocks. The blocks used consist of up to 512 values, comprising a total of at most 32 values from up to 16 rows of the matrix. Conceptually, the $m \times n$ weight matrix could be viewed as being stored as several smaller matrices, each of which being stored in row-major order. Fig. 3.16 shows this for a weight matrix of size 35×66. As can be seen, the weight matrix is divided into smaller blocks, each shown in gray. These smaller blocks are of size 16×32 to the furthest extent possible while still maintaining the layout of the larger matrix.

As for the matrix shown in Fig. 3.16, its width is not divisible by 32 and its height is not divisible by 16. As such, smaller blocks are required to maintain the structure of the larger matrix. This results in some of the data being stored in 16×2 blocks, some in 3×32 blocks and some in 3×2 blocks.

Rearranging the weight data as shown in Fig. 3.16 allows for evaluating the contribution of the input for several rows of the weight matrix without having to load the input multiple times. This reduction in the number of loads of the input stems from improved register usage. The approach relies on loading 32 values from the input and 32 values from each of the 16 rows of the current weight block. The input is multiplied with each of the registers containing the data loaded from the respective rows and the results stored separately. Once this is done, the next 32 values of the input and the respective rows of the weight matrix are loaded. The input data is multiplied with each of the weight registers and the results added to the accumulators for the respective rows. Once the entire input has been used, 16 elements of the vector resulting from the matrix-vector multiplication are obtained by summing the lanes in the respective accumulators. This means that the input vector only has to be loaded from memory once for every 16 rows of the weight matrix.

The approach outlined above describes the case when the weight matrix is of size $16m \times 32n$ for some $m, n \in \mathbb{Z}^+$. In order to achieve generality, weight matrices of any size have to be handled. This means that the implementation must also support blocks that are not of size 16×32, examples of which are shown in Fig. 3.16. While the block sizes may differ, the concepts outlined above may still be used.

	[0]	[1]	[...]	[31]	[32]	[33]	[...]	[63]	[64]	[65]
[0]	0	1	⋯	31	512	513	⋯	543	1024	1025
[1]	32	33	⋯	63	544	545	⋯	575	1026	1027
[⋮]	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮	⋮	⋮
[15]	480	481	⋯	511	992	993	⋯	1023	1054	1055
[16]	1056	1057	⋯	1087	1568	1569	⋯	1599	2080	2081
[17]	1088	1089	⋯	1119	1600	1601	⋯	1631	2082	2083
[⋮]	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮	⋮	⋮
[31]	1536	1537	⋯	1567	2048	2049	⋯	2079	2110	2111
[32]	2112	2113	⋯	2143	2208	2209	⋯	2239	2304	2305
[33]	2144	2145	⋯	2175	2240	2241	⋯	2271	2306	2307
[34]	2176	2177	⋯	2207	2272	2273	⋯	2303	2308	2309

Figure 3.16: Memory layout of a 35×66 weight matrix. Symbols in square brackets indicate row or column indices, remaining symbols are the memory offset of the particular element. The elements are arranged as they would be when viewing the weight matrix written on paper. As an example, the 0^{th} element of row 1 of the matrix is stored as the 32^{nd} value in memory. The gray areas show how the matrix is arranged in memory blocks.

After having computed 16 of the components of the vector resulting from the matrix-vector multiplication, these are added with their corresponding bias terms. More specifically, the computed values are arranged in a vector register and another register is filled with values from the bias vector. These registers are summed, producing 16 terms of the output vector that are written to memory.

The entire algorithm is outlined in Algorithm 3. Here, vector registers are denoted using $v_{...}$ and scalar registers $s_{...}$. The notation $\leftarrow [0]$ means zeroing all lanes of a vector register and $\odot$ is the component-wise multiplication of two vectors. The loop denoted as ***for** is entirely unrolled. For the sake of simplicity, the number of neurons in the output layer is assumed to be a multiple of 16 and the size of the input a multiple of 32.

3.4.1 Special Considerations

One aspect that bears mentioning is how the input is loaded from memory. As the input is read once for every 16 rows of the weight matrix, this constitutes a prime use-case for the circular buffering commonly available in DSPs. This would allow for having the on-chip memory address wrap around to the start of the input array once the entirety of the input has been read. Alternatively, the memory address could be manually reset every iteration of the outer loop in Algorithm 3. Through measuring of the total number of active cycles required for layer evaluation, it was concluded that resetting the pointer manually yielded more efficient code. In the interest of lucidity, this reset is implied rather than shown in Algorithm 3.

Algorithm 3: Evaluation of a fully connected layer.

```
Result: Vector of neuron activations
let v_accs ← array of 16 vector registers
for i ← 0 to NRows/16 do
    for j ← 0 to 16 do
        let v_accs[j] ← [0]
    end
    hwfor j ← 0 to (NCols + 31)/32 do
        let v_input ← Load(input, 32)
        *for k ← 0 to 16 do
            let v_weight ← Load(weights, 32)
            let v_accs[k] ← v_accs[k] + v_input ⊙ v_weight
        end
    end
    for j ← 0 to 16 do
        let s_sum ← Sum(v_accs[j])
        let v_res ← InsertScalar(v_res, j, s_sum)
    end
    let v_bias ← Load(bias, 16)
    let v_res ← v_res + v_bias
    Store(v_res, 16)
end
```

3.5 Added Processor Instructions

Two new instructions were tentatively added to the instruction set of the DSP in order to facilitate evaluation of the machine learning primitives. While neither is a necessity for evaluating the primitives, they both increase the throughput of the implementations.

3.5.1 Variable-Period Packing

The first instruction to be added was one that allowed for easily restructuring the lanes of two vector registers. It was inspired by packing instructions such as packuswb (pack word to byte with unsigned saturation) available in the x86-64 architecture. The packuswb instruction takes two registers containing words, truncates each word to a single byte using unsigned saturation and stores them in order in the destination register. Effectively, this is equivalent to truncating every word to a byte, extracting every other byte of the source registers and storing them in the destination register.

The reason for adding the new pack instruction was to facilitate the extraction of values in periodic lanes as required by the implementations of the convolutional and max-pooling layers. Since the implementations of the layers require support for arbitrary periods, always extracting every other lane did not provide sufficient control. Instead, the instruction had to be made more flexible than packuswb such that it allows for specifying the period. The result was an instruction that accepts two vector registers and an immediate value $p \in \mathbb{Z}^+$ signifying the lane period. The registers are treated as a single compound register. The instruction extracts every p lanes of the compound register and stores the result in the destination register. This means that if $p = 2$, the instruction acts just as packuswb does. This can be seen in Fig. 3.9 in Section 3.2.2.

Issuing the packing instruction with an immediate value $p = 3$, every third lane of the compound source register is extracted. This is shown in Fig. 3.17. A consequence of allowing the programmer to specify the period p is that, assuming n-lane registers, $n \in \mathbb{Z}^+$, the $n - \lceil \frac{2n}{p} \rceil$

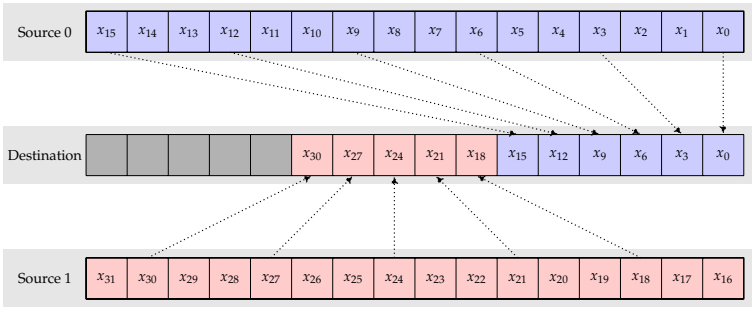


Figure 3.17: Variable packing of two vector registers with period 3. The two source registers are treated as a single 32-lane register and every third lane is extracted and stored in the destination register. The values of gray lanes are unspecified.

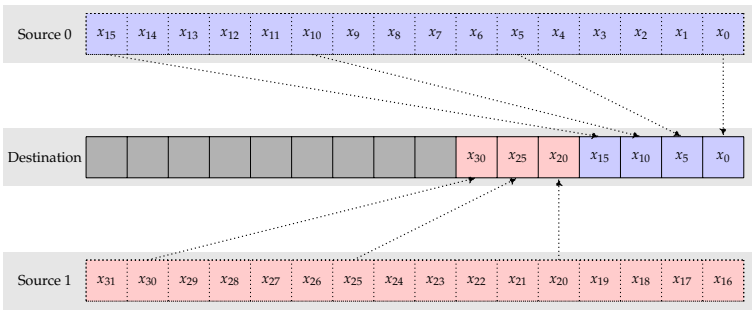


Figure 3.18: Variable packing of two vector registers with period 5.

high lanes of the result register are not written by the instruction. These lanes are shown in gray in Fig. 3.17.

Packing with other choices of period p works analogously to packing with periods 2 and 3. Fig. 3.18 shows the result of specifying the immediate value 5.

3.5.2 Pipelined Multiplication-Addition-Rotation

A significant part of the evaluation of convolutional layers consists of multiplying the lanes of input registers with a scalar filter coefficient. The result is added to accumulator registers and the input registers rotated right after which the procedure is repeated. Using the original instruction set of the processor, this procedure would periodically stall due to the processor having to load new data from memory. In order to do away with this stall, an instruction that allowed for better pipelining the memory load alongside the computations was added.

The added instruction does not expose any functionality that was priorly unavailable but allows for expressing said functionality more concisely. Instead of issuing multiple computational instructions intermixed with separate memory loads, a single instruction suffices. This, in turn, simplifies the work required by the compiler during code generation, resulting in more efficient code.

3.6 Measuring Performance

The thesis uses two different metrics for measuring performance, both of which are presented as percentages. The simplest one is a relative measure of the total number of cycles required to evaluate two separate implementations of a layer. This allows for comparing implementations with respect to which requires the smallest number of cycles to execute.

The second metric measures how well the implementations utilize the hardware units of the DSP. This is done for both a single iteration of the hardware loop of each implementation and for the implementation as a whole. When used in Chapter 4, these are referred to as loop and total usage, respectively. The total usage of a hardware unit is the percentage of the total number of cycles during which the particular unit is active. As an example, a total usage of 40% of the add unit means that the add unit is active 40% of the total number of cycles required to evaluate the layer. The loop usage is defined analogously but instead measures the usage in a single iteration of the hardware loop.

In order to compute the degree to which a particular hardware unit is utilized, the number of instructions that use said unit have to be known. As an example, evaluating the product of multiplying an $m \times n$ matrix with an $n \times 1$ vector requires a total of mn scalar multiplications. Let $k \in \mathbb{Z}^+$ be the number of register lanes the processor is capable of multiplying in a single cycle. Assuming the total number of cycles required for evaluating the product is measured to be some $c \in \mathbb{Z}^+$ cycles, the total hardware usage $u, 0 \leqslant u \leqslant 1$ of the multiplier unit is computed using

$$u = \frac{mn}{kc}. \tag{3.5}$$

Measuring the hardware usage of the multiplier unit during a single iteration of the hardware loop is done analogously to measuring the total usage. More specifically, the loop usage is obtained by measuring the number of cycles in a single iteration of the hardware loop and computing the number of multiplications that are performed in said iteration. The number of cycles is substituted for c in (3.5) and the number of multiplications is used instead of the product in the nominator. In general, the approach can be summarized as

$$\text{Usage} = \frac{\#\text{RelevantInstructions}}{\#\text{Cycles} \times \#\text{RegisterLanes}} \tag{3.6}$$

where an instruction is deemed relevant if it uses the hardware unit for which the usage is to be measured.

4 Results

This chapter presents the results of the thesis. The sections herein are laid out differently as the metrics of success for the respective results they present are distinct. For the memory management described in Section 4.1, the relevant metric is the number of times data elements have to be transferred between on- and off-chip memory. Potential data stalls are also of interest.

In the later sections of the chapter, the results are given mainly in terms of to what degree available hardware resources are used. This effectively serves as a measurement for how well the algorithms are suited to the DSP. How the usage of a hardware unit is computed is detailed in Section 3.6. The results were obtained using a deterministic, cycle-accurate simulator.

4.1 Memory Management

The management techniques described in Section 3.1 allow for evaluating the primitives while transferring each data element from off- to on-chip memory only once.

The asynchronous buffering used makes it possible to issue data transfers while simultaneously performing calculations. Despite this, there are two points where data stalls are guaranteed. One is when execution starts, as data has to be available in the on-chip memory before the computations can be performed. Similarly, no computations are performed while the last batch of output data is written. This is due to there being no computations left to perform at this point.

Restructuring the weight data used for the fully connected layer as described in Section 3.4 reduces the number of times the elements of the input vector have to be loaded by roughly 94%. This was found to reduce the number of data stalls while evaluating the fully connected layer, effectively doubling the usage of add and multiplier units.

4.2 Convolutional Layer

Since the evaluation of a convolutional layer requires additions, multiplications and restructuring of data, the implementation makes use of the add, multiplier and permuting units of the DSP. Although the method presented in Section 3.2.2 supports an arbitrary stride, the

Table 4.1: Hardware usage when evaluating a convolutional layer using a 3×3 kernel with different hyperparameters. The percentages are presented as total/loop usage. Results for hyperparameter choices for which the convolution is undefined are marked with N/A.

		Padding			
		0	1	2	3
Stride	1	45%/75%	40%/100%	43%/60%	33%/38%
	3	10%/20%	N/A	N/A	8%/18%

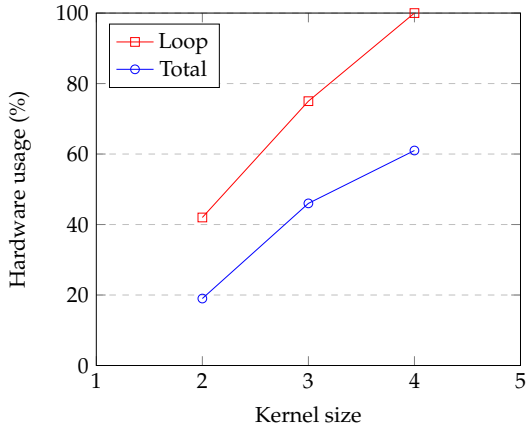


Figure 4.1: Degree of usage, both in total and in the hardware loop, of the add and multiplier units as a function of kernel size. The convolution is computed using a square kernel, meaning a kernel size 2 signifies that a 2×2 kernel is used.

implementation assumes the stride is either 1 or the same as the width of the filter kernel used. This section presents the degree to which the add and multiplier units are used in the proposed implementation. As for the permuting unit, no reliable way of measuring its usage when evaluating the convolutional layer was found.

Table 4.1 shows the degree to which the add and multiplier units are used when convolving a 228×228 image with a kernel of size 3×3 with different choices of hyperparameters. The first percentage presented for each cell in the table is the total usage and the second the loop usage. By definition, it is not possible to convolve a 228×228 image with a 3×3 kernel using a stride 3 with a padding of either 1 or 2. These configurations are marked as N/A. Since the evaluation of a convolutional layer requires the exact same number of multiplications and additions, the add and multiplier units are active to the same degree. Considering this, Table 4.1 presents the usages of the add and multiplier units for a particular configuration as a single pair of percentages. As an example, the pair 45%/75% found in the first cell means that the add and multiplier units each are active 45% of the total cycles and 75% of the loop cycles.

Fig. 4.1 shows how the degree to which the hardware is used varies with the size of the filter kernel. The measurements were obtained using an unpadded 228×228 image, applying the kernels with a stride of 1. As in Table 4.1, the percentages presented are the separate usages of both the add and multiplier units.

4.3 Max-Pooling Layer

The max-pooling layer requires comparisons and restructuring of data, meaning the add and permuting units are the two most active. This section presents the usage of these units when evaluating max-pooling layers with different kernel sizes. The results were obtained using an

Table 4.2: Hardware usage when evaluating a max-pooling layer using kernels of different sizes. The percentages are presented as total/loop usage.

	Kernel Size			
	2×2	3×3	4×4	6×6
Add Unit	25%/50%	16%/40%	4%/68%	3%/67%
Permuting Unit	25%/100%	12%/60%	4%/100%	2%/67%

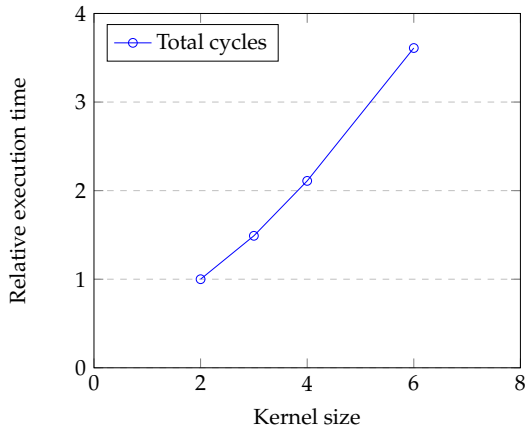


Figure 4.2: Relative execution time of the entirety of the max-pooling implementation as a function of kernel size. Pooling with a 2×2 kernel is used as the baseline.

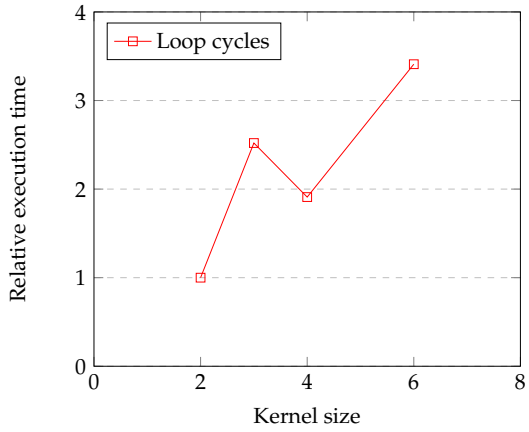


Figure 4.3: Relative execution time of evaluating a single iteration of a hardware loop in the max-pooling implementation. The execution time is a function of kernel size. Pooling with a 2×2 kernel is used as the baseline.

input image of size 228×228. The kernel sizes tested are 2×2, 3×3, 4×4 and 6×6. A kernel of size 5×5 is not considered as it is not conducive to the chosen input size. The degree to which the add and permuting units are used while evaluating a max-pooling layer with the above kernel sizes is shown in Table 4.2.

Fig. 4.2 shows the relative execution time for evaluating an entire max-pooling layer with kernels of size 2×2, 3×3, 4×4 and 6×6. The application of the 2×2 kernel is used as the baseline. Fig. 4.3 shows the relative execution time of a single iteration of the hardware loop under the same circumstances.

For none of the results reported in this section was the memory transfer limited by a lack of capacity.

Table 4.3: Hardware usage when evaluating a fully connected layer with different input and output sizes. The precentages are presented as total/loop usage.

		Output size		
		48	100	256
Input size	256	25%/94%	31%/94%	37%/94%
	496	39%/94%	47%/94%	52%/94%

4.4 Fully Connected Layer

The fully connected layer is evaluated as a series of MAC operations for which the add and multiplier units are the most relevant. The usage of these units when evaluating fully connected layers with different input and output sizes are given in Table 4.3. As is the case with the convolutional layer, the add and multiplier units were found to be active to the exact same degree also in the fully connected layer. As a result, the percentages given in Table 4.3 are the usages of both the add and multiplier units.

4.5 Added Processor Instructions

This section presents the improvements observed as a result of adding and using the instructions proposed in Section 3.5. The results are presented mainly as a relative measure of the number of cycles required to evaluate particular layers. For the pipelining instruction, the difference in hardware usage observed when using the instruction is also provided.

4.5.1 Variable-Period Packing

The variable period packing is used in the implementations of the convolutional and max-pooling layers. In the max-pooling implementation, the effectiveness of the instruction differs depending on the kernel size. For a 3×3 kernel, the number of cycles required to evaluate the layer is reduced by roughly 6%. The effects are lessened by using a larger size kernel as this means that the percentage of the active cycles that are used for restructuring the data is smaller. As a result, only a 0.4% reduction in active cycles is observed when using a 7×7 kernel.

In the implementation of the convolutional layer, the restructuring of the data constitutes a negligible percentage of the required work. As such, the added instruction does little to improve performance. Here, the main benefit of adding the instruction is instead that it simplifies the implementation.

4.5.2 Pipelined Multiplication-Addition-Rotation

The instruction that allows for better pipelining of multiplications, additions and rotations intermixed with memory loads is used in the implementation of the convolutional layer only. The reduction of total cycles required to evaluate a convolutional layer differs depending on the choice of hyperparameters. Table 4.4 shows the total reduction in cycles observed for different choices of hyperparameters when convolving a 228×228 image with a 3×3 kernel.

Fig. 4.4 shows the hardware usage when convolving a 228×228 image using different size kernels without the instruction improving the pipelining. The hardware usage observed when using the pipelining instruction can be seen in Fig 4.1. Fig. 4.5 shows the gain in hardware usage obtained by using the pipelining instruction. In other words, it shows the differences between the respective data points in Fig. 4.1 and Fig. 4.4.

Table 4.4: Reduction in total cycles resulting from the use of the instruction performing pipelined multiplications, additions and rotations. Results for hyperparameter choices for which the convolution is undefined are marked with N/A.

		Padding			
		0	1	2	3
Stride	1	23%	13%	22%	20%
	3	44%	N/A	N/A	18%

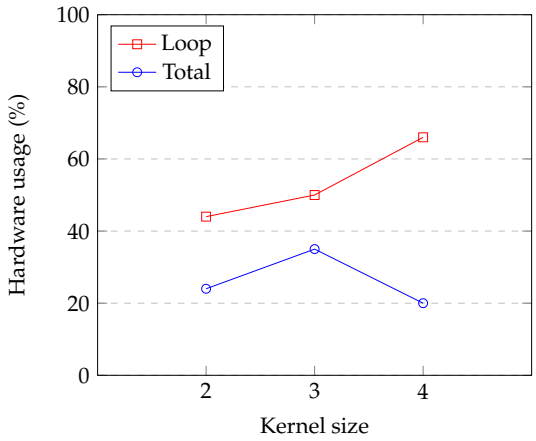


Figure 4.4: Hardware usage when evaluating a convolutional layer without the improved pipelining.

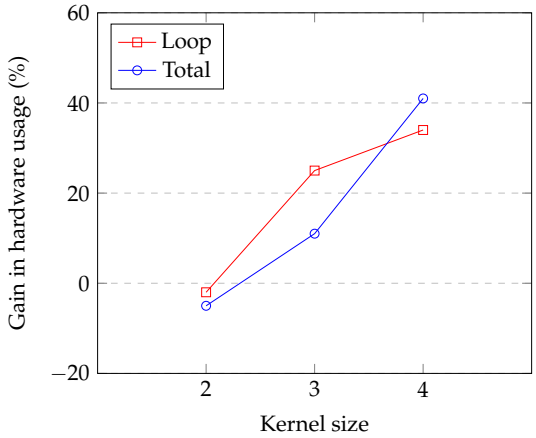


Figure 4.5: The gain in hardware usage obtained by using the added pipelining instruction.

5 Discussion

This chapter discusses the results and method of the thesis. Hypothetical ways it may come to affect society are also addressed.

5.1 Results

The results are promising albeit far from optimal. This section addresses the memory management, the different primitives and the added instructions separately.

5.1.1 Memory Management

While copying each data element from off- to on-chip memory only once is theoretically optimal, there are still potential caveats that should be discussed. Firstly, the actual memory management does not come without cost. In order to ensure that data transfers are issued at the correct time, several memory addresses must be kept track of. While the operations themselves are not particularly expensive, the added complexity may affect code generation. This, in turn, may result in less optimal code being generated, increasing the execution time.

Another potential memory issue is encountered when working with large inputs. As the on-chip memory is small, large inputs may require a large number of memory transfers from off- to on-chip memory. The asynchronous buffering used relies on the computations performed to hide the transfer latency. If the input width is sufficiently large, there is not enough data available to perform the number of computations required to hide the transfer latency. Instead, data transfers would have to be issued each iteration, something that would lead to a large number of data stalls. The results are even worse for input sizes large enough that they do not fit in the on-chip memory. When this is the case, the approaches proposed in the thesis would no longer produce the correct results.

Lastly, the choice of using blocks of 16 rows when restructuring the weight data for the fully connected layer should be discussed. The choice of block size means that the elements of the input vector have to be loaded once every 16 rows of the weight matrix. In theory, it would be possible to load the input elements only once. Assuming the weight matrix is of dimensions $m \times n$, this would be accomplished by working with blocks of size $m \times 32$ instead of 16×32. In practice, this is limited by the hardware. The restructuring of the data is performed in order to reduce the number of memory loads. The success of the restructuring

approach hinges on all data worked on at a particular point being kept in registers. Using blocks of size $m \times 32$, this would be possible only for small choices of m and would thus lead to a loss of generality. As such, the choice of blocks comprising 16 rows was considered a good compromise between theoretical optimality and hardware restrictions.

5.1.2 Convolutional Layer

As is evident from the results presented in Section 4.2, the implementation of the convolutional layer fits the DSP well provided that the stride is 1 and the padding is small. Table 4.1 shows that it is possible to utilize the add and multiplier units optimally while evaluating convolutional layers with certain parameter configurations. This is not surprising as the evaluation of a convolutional layer is performed by essentially applying a two-dimensional FIR filter. Applying one-dimensional FIR filters is a common use case for DSPs. As such, it was expected that the processor would handle the generalization to two dimensions well.

Unfortunately, the degree to which the hardware is utilized fluctuates heavily with the choice of hyperparameters. This can be seen in Table 4.1. Furthermore, Fig. 4.1 shows that the usage varies heavily also with the choice of kernel size. While the results seem to indicate that a larger-size kernel makes better use of the available hardware, this cannot necessarily be concluded from the results presented here alone.

Table 4.1 shows that using a stride larger than one leads to abysmal hardware usage. This is not surprising as the implementation of strided convolution performs a number of superfluous computations. As mentioned, it issues the same instructions as when evaluating convolution with stride 1 and then discards the unwanted results.

It should be noted that the results measure only the cycles that contribute to the actual output. Due to this, a larger stride results in a lower hardware usage since it causes the number of wasted cycles to increase. As the usages for strided convolution presented in Table 4.1 are obtained using a stride 3, the add and multiplier units are actually active during roughly 60% of the loop cycles. However, since only every third value is used for the output, the active usage is no more than 20%. As a result, the drastic reduction in loop usage when comparing the strides in Table 4.1 is explained largely through how inefficiently the implementation handles the stride.

Table 4.1 shows that the effect padding the image has on the hardware usage is unpredictable. Logically, the hardware usage in the loop should be at its peak when using no padding. This is due to the emulation of the padding requiring the issuing of additional instructions. As can be seen in Table 4.1, this does not appear to be the most efficient configuration. Instead, the loop usage is higher when using a padding of 1. This is likely a consequence of the implementation being ill-suited to the processor, something that is discussed further in Section 5.2.2.

As can be seen in Table 4.1, the implementation of strided and padded convolution is the least efficient in terms of hardware usage. This is at least partially in line with what has been observed thus far and comes as no surprise as the complexity of the code increases with the number of hyperparameters that need to be supported.

5.1.3 Max-Pooling Layer

The max-pooling layer is the one least suited to the DSP. The main reason for this is the relatively small number of instructions that are issued for each input element. This claim is corroborated by Table 4.2. As can be seen, the total usage of relevant hardware units rarely exceeds 25%. This means that a large part of the processing power of the processor is unused during evaluation. Despite this, the hardware usage in the hardware loop is relatively high. The contrast is especially apparent when considering the hardware usage of the permuting unit when applying a 4×4 kernel. The substantial difference in total and loop usage indicates that the vast majority of the work performed during evaluation takes place outside the loop.

The exact reason for the low degree of hardware usage is not immediately apparent. As Table 4.2 shows, the usage of the most heavily used hardware units is capped at 100% only when the kernel is of size 2×2 or 4×4. In the remaining cases, there is an excess of computational resources available. As noted in Section 4.3, the memory transfer capacity was not the limiting factor either. Instead, the most probable cause is the sequentiality of the algorithm used. Looking at the implementation, each instruction requires the previous instructions to have finished. As an example, the final packing of the data cannot be performed until all the max computations are finished. This means that the degree of pipelining that can be performed by the processor is limited. The limited pipelining would, in turn, result in the lower hardware usages observed and, by extensions, longer execution times. A potential way to address this issue would be to unroll the hardware loop. This might allow for better pipelining of the instructions, meaning the hardware would be used to a larger degree.

As can be seen in Fig. 4.2, the number of cycles required to evaluate the layer appears to grow linearly with the size of the kernel. This is expected as evaluating a max-pooling layer with an $m \times m$ kernel requires simultaneous access to m rows. This increases the total number of memory loads which, along with the additional computations required, explains the increase in execution time.

Fig. 4.3 shows that a kernel size of 3×3 breaks the mould in terms of required loop cycles. Despite this, the total number of cycles in Fig. 4.2 follows the overall trend. The reason for this is that it was found to be more efficient to work with an even number of elements in each iteration of the hardware loop. As a result, the loop body itself is twice the size of the bodies of the loops used for kernels of even dimensions. This, in turn, means that the total number of loop iterations is halved. The value shown for the 3×3 kernel in Fig. 4.3 is the number of cycles required for one iteration of the unrolled loop. This explains how the number of cycles required for a single iteration deviates to such an extent while the total number of cycles follows the trend.

5.1.4 Fully Connected Layer

The implementation of the fully connected layer is well-suited to the DSP as can be seen in Table 4.3. It can be observed that the total hardware usage increases with the size of the input and output. This is expected as working with larger inputs and outputs means that a larger percentage of the active cycles is spent in hardware loops performing multiplications and additions. This, in turn, activates the add and multiplier units to a larger degree. While the total usage increases with the size, the loop usage remains constant across the different configurations. This is due to the body of the loop remaining the same regardless of input and output sizes.

The results suggest that it should be theoretically possible to achieve a high degree of total hardware usage. All that would need to be done is to evaluate fully connected layers that are sufficiently large. In practice, it is likely that this would be limited by available on-chip memory before a very high degree of hardware usage is achieved. Nevertheless, the results show that by providing a sufficiently large amount of on-chip memory, the algorithm proposed in the thesis might be able to achieve near-optimal hardware usage.

5.1.5 Added Processor Instructions

As the results show, the variable-period packing instruction reduces the number of cycles required to evaluate max-pooling layers. The impact of adding the instruction is lesser the larger the kernel used. As mentioned, this is no surprise as using a larger kernel means that a larger percentage of the active cycles is used to perform the max computation. This, in turn, implies that the relative portion where the data is restructured is smaller.

The reduced efficiency observed for larger kernels notwithstanding, the instruction does decrease the number of active cycles for evaluating max-pooling layers, regardless of kernel

size. Considering the fact that the improvement observed for smaller kernels is more than 5%, the performance increase alone merits that it be considered an alternative. As the instruction was found to also greatly simplify the implementation of both the convolutional and max-pooling layers, it should be a good candidate for bolstering the instruction set of the DSP.

The most important instruction to include if wanting to evaluate machine learning primitives on a DSP is the pipelining instruction. As is evident in Table 4.4, it was found to reduce the total number of cycles required for evaluating convolutional layers by up to 44%. Without said instruction, the hardware usages would be nowhere near the final tallies presented in Section 4.2. The instruction is, however, not a panacea. As shown in Fig. 4.5, using the instruction does not always yield an improvement. Instead, the hardware usage, both in the loop and in total, is reduced when using the instruction in the evaluation of a convolutional layer using a 2×2 kernel. This is, again, likely in large part due to the code not being well-suited to the processor as discussed in Section 5.2.2.

5.2 Method

There are a few points that should be made with regards to the method of the thesis. Firstly, rather strong assumptions are made in terms of memory requirements. It could be argued that assuming that an entire row of data fits in the on-chip memory is a necessary prerequisite. There is, however, no guarantee that an actual neural network would adhere to this assumption. As a result, there is likely to exist neural networks for which the algorithms described in the thesis simply would not work. While it may be possible to make some of these networks evaluable on the DSP by using neural network pruning, this is not guaranteed.

It would be theoretically possible to design algorithms for layers of arbitrary sizes. This would, however, require a large number of memory transfers. As a result, evaluating them would likely be slow. Due to this, the approach used in the thesis favors more efficient implementations that work for most, albeit not necessarily all, cases.

A significant source of inaccuracy is that different DSPs have different memory architectures. As a result, an implementation that works for the DSP used in the thesis is not guaranteed to be applicable to DSPs in general. This highlights an important limitation in the thesis, namely that it works with only a single DSP. Working with multiple processors would have been nigh-on impossible due to reasons including hardware availability and time constraints. This does, however, not change that fact that the narrow choice of hardware lessens the generality of the thesis. More concretely, it is not guaranteed that the results found in the thesis are replicable using other DSPs. This may be due to a large number of reasons including differing instruction sets, memory hierarchies and development tools. When working with the DSP used throughout the thesis, the results should, however, be both reliable and possible to replicate.

5.2.1 Convolutional Layer

The implementation of the convolutional layer in particular suffers from a few notable shortcomings. Firstly, the approach described for achieving strided convolution computing excess output elements and discarding them is vastly suboptimal. This is especially obvious when considering the results in Table 4.1. Unfortunately, no practical way of avoiding this was found during the thesis. The reason for this is that if convolution with a kernel of width $n \in \mathbb{Z}^+, n > 1$ is evaluated, producing each element of the output requires n input elements from each row under the kernel. Using the memory access pattern described in Section 3.1, this means that each output element requires data from n adjacent lanes in the vector register. In other words, assuming one-dimensional convolution, the i^{th} output element $y_i \in \mathbb{R}$ is

computed from input samples $x_j \in \mathbb{R}$ using

$$y_i = \sum_{j=in}^{(i+1)n-1} c_j x_j \tag{5.1}$$

where $c_j \in \mathbb{R}$ are the filter coefficients. This shows that, in the general case, each lane of the vector register is required in order to produce the strided output. Since vector processing performs the same operation on each lane, it is therefore unavoidable that the unwanted elements are computed. The one case where computations could be avoided is when the stride is larger than the horizontal extent of the filter kernel. In this case, data that is never used would be loaded into the registers. This case does, however, seem unlikely as it would disregard portions of the input data completely, begging the question as to why these elements were there to begin with.

It would be theoretically possible to avoid computing the excess output elements of strided convolution by leveraging strided memory loads. Assuming an $m \times n$ filter kernel is to be applied with a stride $s \in \mathbb{Z}^+$ and that the vector registers used comprise $k \in \mathbb{Z}^+$ lanes, the approach would rely on issuing a total of n memory loads per row, each with stride s. The first load would be performed with an offset of 0, meaning that the contents of register $\mathbf{v}_0$ after the load would be given by

$$\mathbf{v}_0 = \begin{bmatrix} w_0 & w_s & w_{2s} & \dots & w_{(k-1)s} \end{bmatrix} \tag{5.2}$$

where w_j is the j^{th} input element, zero indexed. The subsequent load would be offset by a single element, meaning the contents of register $\mathbf{v}_1$ would be

$$\mathbf{v}_1 = \begin{bmatrix} w_1 & w_{s+1} & w_{2s+1} & \dots & w_{(k-1)s+1} \end{bmatrix}. \tag{5.3}$$

Equations (5.2) and (5.3) show the contents of the registers when loading data for the first k filter applications for a particular row. As the input may be wider than k pixels, an additional offset must be taken into account. In general terms, the contents of the i^{th} register $\mathbf{v}_i$ would be given by

$$\mathbf{v}_i = \begin{bmatrix} w_{i+b} & w_{s+i+b} & w_{2s+i+b} & \dots & w_{(k-1)s+i+b} \end{bmatrix} \tag{5.4}$$

where $b \in \mathbb{Z}^+$ is the offset of the first element to be loaded into register $\mathbf{v}_0$. Data would be loaded using this approach for a total of n times after which all values required to evaluate k filter applications for a particular row under the filter kernel have been loaded. The lanes of the respective vector registers $\mathbf{v}_i$, $i = 0, 1, \dots, n-1$ would then be multiplied with the i^{th} filter coefficient for the row in question. After this, summing the n registers would produce a total of k contributions from the row being processed. The procedure would suitably be implemented as $n-1$ separate steps, requiring only two vector registers.

By computing the contributions of all m rows under the filter kernel using the strided load approach and summing them, a total of k output elements of the strided convolution would be produced. This approach computes no superfluous output elements, making it theoretically superior to the one detailed in the thesis. The approach is, however, not without its flaws. The most obvious one is that when computing the contribution of an $m \times n$ kernel, it issues a total of mn memory loads. In contrast, the approach used in the thesis requires only m loads for computing the contribution of a kernel of the same size. As such, the strided load approach is more likely to suffer from data stalls, especially for large filter kernels. Additionally, the strided load approach would have to load some elements multiple times since it is unlikely that the processor provides enough registers to keep data persistent across iterations as proposed in Section 3.2.2. As such, it is not guaranteed that the strided load approach would lead to an increase in hardware usage. While it would have been interesting to compare the two approaches, limitations in the hardware used in the thesis made it impossible to implement the alternative algorithm.

A flaw in the evaluation of horizontally padded convolution is that even though the padding consists of only zeroes, these zeroes are still used for computations. A more sophisticated approach would have noted that no matter what the weights of the filter kernel are, the padded lanes will contribute nothing to the numerical value of the convolved feature. As such, there is theoretically no need to perform any multiplications or additions for the padded lanes. The implementation of this approach is, much like in the case of the strided convolution, foiled by the use of vector processing. When issuing a packed multiplication, all lanes of the vector register are multiplied regardless of their content. Due to this, there is no way of avoiding the multiplication of the padded lanes, despite its contribution being non-existent.

It should be noted that the approach used to emulate zero padding would generalize poorly to padding techniques involving edge pixels. While modify the proposed method for vertical padding such that the edge pixels are repeated should prove simple, repeating horizontal pixels would be more expensive. A straightforward way of emulating horizontal padding where edge pixels are repeated would be to simply extract the edge pixels from the vector registers and insert them in the lanes reserved for the padding. An alternative approach would instead use shifts and bitmasks. Both options rely on the DSP used providing the required instructions, something that is not guaranteed. Additionally, as the alternatives require the issuing of a larger number of instructions than the proposed padding implementation, their performance would likely be worse. Using mean values of edge pixels would presumably be even less efficient as reduction operations typically lend themselves poorly to vector processing.

Another issue with the implementation of the convolutional layer pertains to higher-order convolution. The approach proposed in the thesis evaluates a three-dimensional convolution as several two-dimensional ones and sums the results. In practice, it is unlikely that the output of all depth slices can be held in the on-chip memory during evaluation. As a result, this requires evaluating each depth slice individually and temporarily writing the result back to the off-chip memory. Once all depth slices have been processed, the output data from the slices would again have to be read into the on-chip memory and summed, producing the layer output. As is evident, this performs a large number of unwanted data transfers between on- and off-chip memory.

Ideally, the depth slices of three-dimensional convolution would all be evaluated at the same time. This could be done by merging the separate evaluations of the depth slices into a single one. Instead of computing the entire two-dimensional convolution of one slice and then moving on to the next, one kernel application of each slice would be computed. This would allow for accumulating the results of the depth slices in a single register, meaning no extra memory transfers would have to be made. During the thesis, attempts were made to use this approach but it was found impossible to realize due to hardware restrictions. Nevertheless, assuming the hardware supports it, this approach would be theoretically superior to the one proposed in the thesis.

5.2.2 Consequences of Using Metaprogramming

The implementations rely heavily on compile-time code optimization using metaprogramming. As the code was written in C++, this was achieved through the use of both constexpr and template metaprogramming. The choice of using metaprogramming was made in order to support a wider variety of layer configurations.

DSPs are complex processors. Due to this, it is hard to develop compilers that are able to consistently generate efficient code, especially when using templates. As a result, DSP codebases usually contain separate implementations for each distinct parameter configuration. Not only does this ease the burden on the compiler, it also allows for specific optimizations that are not possible when writing generic code.

In the context of the thesis, not using generics would have required separate implementations for every single combination of layer parameters such as kernel size, stride, padding and input size. As such, it would have been difficult to produce the required implementations within the allotted time. Despite this, the fact remains that the code instantiated from the templates is potentially far less efficient than separate implementations would have been. This might, at least in part, further explain the wild fluctuations in the results obtained.

Another aspect to consider resulting from the use of metaprogramming is the increased code size. Supporting arbitrary input sizes requires additional loops. In order to control which loops are evaluated as hardware loops, some of these additional loops have to be unrolled completely. It was found that the unrolling functionality exposed by the compiler was not reliable enough for this. Instead, the unrolling was achieved using compile-time recursion where each function call was force-inlined. This means that the code that is instantiated from the templates may be large, likely resulting in poor use of the instruction cache of the DSP.

5.2.3 Source Criticism

The sources used in the thesis are either research papers, textbooks, reference manuals or compiler documentation. While both reference manuals and compiler documentation should be reliable, the research papers and textbooks may be discussed further.

The majority of sources used in the thesis are research papers. Of these, in turn, the vast majority were presented during proceedings or published in journals that use peer-review. While this is not sufficient to guarantee the quality of the papers, it is an indication of them adhering to a certain standard.

Two research papers, [7] by Rabanser, Shchur and Günnemann and [13] by Derksen, Friedland, Lim and Wang are, as of the time of writing, published only on arXiv. The latter is an archive that does not require peer-review. The authors of [7] claim that the paper is a working draft and that it as such has yet to be published in a peer-reviewed journal. Additionally, the co-author Günnemann has well over 1000 citations to his name. As for [13], Derksen and Friedland alone boast a combined 10 000 citations throughout their careers according to Google Scholar. As such, the lack of peer-review is mitigated by the merit of the authors.

Regardless of whether the research papers used were peer-reviewed or not, the content was examined critically before citation. As such, the primary sources used for the thesis are considered of satisfactory reliability.

As for the textbooks used, the ones detailing DSPs should be discussed. The main aspect to be criticised is the fact that the majority were written more than ten years prior to the thesis. The lack of recency is not ideal. The reason these books were used is the apparent lack of literature detailing DSPs. In other words, they were the most recent books found that explain the general architecture sufficiently well. As they were the best resources available, their content was considered adequate for the thesis.

5.3 The Work in a Wider Context

The contents of the thesis are of limited direct societal impact. While the use of artificial intelligence is far from devoid of controversy and ethical dilemmas, the adaptation of machine learning algorithms to a specific processor should not directly contribute to these.

It could be argued that by facilitating evaluation of machine learning primitives on a DSP, the thesis might contribute to reducing the cost of handheld devices. This, in turn, might make the use of such devices more prevalent in poorer countries. While this may enrich the everyday lives of the people in these countries, it would also subject them to the aforementioned ethical dilemmas that accompany artificial intelligence. The gravity of this prospect is lessened by the fact that the step from a theoretical possibility proposed in a thesis to full-scale

worldwide development is enormous. It could also be argued that as the popular sentiment seems to embrace artificial intelligence, these dilemmas have to be addressed sooner rather than later regardless.

Another potential consequence of hypothetically contributing to cheaper handheld devices is that the population in wealthier countries might begin to treat smartphones and similar as disposable. The disposing of devices might, in turn, have negative environmental consequences. While it seems unlikely the thesis will gain traction enough for this to occur, the eventuality should be taken seriously none the less.

6 Conclusion

The thesis proposes DSP implementations of the machine learning primitives convolutional, max-pooling and fully connected layers that all utilize vector processing. The results show that convolutional and fully connected layers may be implemented efficiently on the DSP whereas max-pooling makes less use of available hardware. A notable exception to the above is when evaluating convolutional layers with a stride larger than 1 as this is handled poorly by the proposed implementation.

The large amount of data used by neural networks is proposedly handled by transferring it to the on-chip memory in parts. Once a piece of data has been loaded into on-chip memory, all computations that require the data in question should be performed before it is evicted. This allows for transferring data only once, thereby minimizing the time wasted on data transfers. Asynchronous buffering may be used to largely hide the transfer latency that occurs as a result of required data transfers.

While the data is still in the on-chip memory, loading it multiple times is of little impact in terms of execution time. Despite this, the number of loads from on-chip memory should be kept to a minimum. This is suitably achieved by storing data in unused registers whenever possible. Restructuring the data may also be significant as shown by the results detailing the implementation of the fully connected layer.

In spite of the memory management techniques described, there do exist neural networks for which constructing efficient implementations on DSPs seems unfeasible. This is especially true for networks with layers so large that enough data to evaluate an entire row of the output does not fit in the on-chip memory of the DSP.

In order to better adapt the DSP to evaluating machine learning primitives, an instruction that allows for efficiently restructuring the data in register lanes should be provided. Additionally, instructions that allow the processor to predictably pipeline memory loads, multiplications, additions and rotations may be of interest. During the thesis, the addition of such instructions was found to result in performance increases of up to 44%.

6.1 Future Work

There are multiple ways the results could be improved further. One key aspect would be to evaluate the approach proposed in the thesis on other DSP architectures. This would give an

indication of how well the algorithms generalize to DSPs and whether the latter are indeed an alternative for evaluating machine learning primitives.

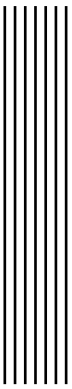
By far the least efficient implementation proposed in the thesis is that for evaluating strided convolution. As the algorithm used is clearly wanting, finding an approach that makes better use of the hardware would be most beneficial in terms of improving the results. A specific approach that would be interesting to explore is the one achieving strided convolution by relying on strided memory loads described in Section 5.2.1.

It would be interesting to investigate whether the results could be improved by writing non-generic implementations of the algorithms. While such implementations would likely be cumbersome to produce, it is expected that they would result in more efficient code being generated by the compiler. This, in turn, might make the implementations behave more predictably.

Another way of potentially improving performance would be to prune the layers of the network as described in Section 2.2.1. This might allow for reducing the number of computations required, thereby making it possible to evaluate the respective layers in fewer cycles. It would also be interesting to investigate how pruning affects hardware usage. While the results shown in Fig. 4.1 and Table 4.3 indicate that it may have negative effects on the hardware usage, such a conclusion cannot be drawn from the results obtained in the thesis alone.

There are a number of alternative approaches for computing convolution whose impacts would be interesting to investigate. Among these are Fourier transform-based and separable convolution, described in Sections 2.1.2.1 and 2.1.2.2, respectively. These were not considered throughout the thesis due to concerns regarding overall performance and generality. Despite this, they may outperform the proposed implementations under certain circumstances.

While the algorithms proposed have been developed over an extensive period of time, their performance could undoubtedly be improved through further tweaking. Additionally, if evaluation of machine learning primitives on a DSP is to be pursued, changes to the DSP should be considered as well. The most conspicuous change would be to increase the amount of on-chip memory available in order to facilitate the evaluation of larger layers. Before making such changes, a large number of factors would have to be considered. Among these are the consequences increased memory sizes may have for the primary use cases of the DSP, the added production costs and the increased energy consumption.



Bibliography

[1] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. "AI Benchmark: Running Deep Neural Networks on Android Smartphones". In: *The European Conference on Computer Vision (ECCV) Workshops* 11133 (Sept. 2018), pp. 288–314. DOI: 10.1007/978-3-030-11021-5_19.

[2] Raushan Kumar, Sourabh Kumar, and Yogesh Rana. "The Role of Digital Signal Processors (DSP) for 4G Mobile Communication Systems". In: *Interational Journal of Electrical and Electronics Research* 2 (3 Sept. 2014), pp. 235–238. ISSN: 2348-6988.

[3] Steven W. Smith. *The Scientist and Engineers Guide to Digital Signal Processing*. 2nd ed. San Diego, CA: California Technical Pub., 1999. ISBN: 9780966017663.

[4] Edmund Lai. *Practical Digital Signal Processing for Engineers and Technicians*. 1st ed. Amsterdam, the Netherlands: Newnes, 2005. ISBN: 9780750657983.

[5] Michael Parker. *Digital Signal Processing: Everything You Need to Know to Get Started*. Boston, MA: Newsnes, 2010. ISBN: 9781856179218.

[6] Joseph Kolecki. "An Introduction to Tensors for Students of Physics and Engineering". In: *NTRS* (Oct. 2002). NTRS: 2002 – 211716. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20020083040.pdf.

[7] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. "Introduction to Tensor Decompositions and Their Applications in Machine Learning". In: *arXiv* (Nov. 2017). arXiv: 1711.10781. URL: https://arxiv.org/abs/1711.10781.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. ISBN: 9780262035613.

[9] Debesh Choudhury. "Teaching the Concept of Convolution and Correlation using Fourier Transform". In: *14th Conference on Education and Training in Optics and Photonics: ETOP 2017*. Ed. by Xu Liu and Xi-Cheng Zhang. Vol. 10452. International Society for Optics and Photonics. SPIE, 2017, pp. 183–188. DOI: 10.1117/12.2267976. URL: https://doi.org/10.1117/12.2267976.

[10] Pavel Karas and David Svoboda. *Algorithms for Efficient Computation of Convolution*. Rijeka, Croatia: INTECH, Jan. 2013. ISBN: 9789535108740. DOI: 10.5772/51942. URL: https://www.intechopen.com/books/design-and-architectures-for-digital-signal-processing/algorithms-for-efficient-computation-of-convolution.

[11] Tahmid Abtahi, Amey Kulkarni, and Tinoosh Mohsenin. "Accelerating Convolutional Neural Network with FFT on Tiny Cores". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)* (Baltimore Marriott Waterfron, Baltimore, May 28, 2017–May 31, 2017). IEEE, May 2017, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050588.

[12] Raf Vandebril, Nicola Mastronardi, and Marc van Barel. *Matrix Computations and Semiseparable Matrices, Volume 1: Linear Systems*. Vol. 1. Baltimore, MD: Johns Hopkins University Press, 2008. ISBN: 9780801887147.

[13] Harm Derksen, Shmuel Friedland, Lek-Heng Lim, and Li Wang. "Theoretical and Computational Aspects of Entanglement". In: *arXiv* (May 2017). arXiv: 1705.07160. URL: https://arxiv.org/abs/1705.07160.

[14] Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. "Fast Matrix Rank Algorithms and Applications". In: *Proceedings of the Fourty-Fourth Annual ACM Symposium on Theory of Computing* (May 2012), pp. 546–562. DOI: 10.1145/2213977.2214028.

[15] Shmuel Friedland and Lek-Heng Lim. "Computational Complexity of Tensor Nuclear Norm". In: *Mathematics of Computation* 87.311 (2018), pp. 1255–1281. DOI: 10.1090/mcom/3239.

[16] Dave Hale. *An Efficient Method for Computing Local Cross-Correlations of Multi-Dimensional Signals*. CWP report 544. 2006.

[17] Christopher M. Bishop. *Pattern Recognition and Mechine Learning*. 1st ed. Berlin Heidelberg, Germany: Springer-Verlag, 2006. ISBN: 9780387310732.

[18] Rikiya Yamashita, Mizuho Nishio, Richard Do, and Kaori Togashi. "Convolutional Neural Networks: An Overview and Application in Radiology". In: *Insights into Imaging* 9 (June 2018). DOI: 10.1007/s13244-018-0639-9.

[19] Yann Le Cun, John S. Denker, and Sara A. Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1990, pp. 598–605. ISBN: 1558601007. DOI: 10.5555/109230.109298.

[20] Renjie Xie, Heikki Huttunen, Shouxin Lin, Shuvra S. Bhattacharyya, and Jarmo Takala. "Resource-Constrained Implementation and Optimization of a Deep Neural Network for Vehicle Classification". In: *2016 24th European Signal Processing Conference (EUSIPCO)* (Aug. 28, 2016–Sept. 2, 2016). Budapest, Hungary: IEEE, 2016, pp. 1862–1866. ISBN: 9780992862657. DOI: 10.1109/EUSIPCO.2016.7760571.

[21] M. Gethsiyal Augasta and Thangairulappan Kathirvalavakumar. "Pruning Algorithms of Neural Networks — a Comparative Study". In: *Central European Journal of Computer Science* 3.3 (Sept. 2013), pp. 105–115. ISSN: 2081-9935. DOI: 10.2478/s13537-013-0109-x.

[22] Saad Albawi, Tareq Abed Mohammed, and Saad ALZAWI. "Understanding of a Convolutional Neural Network". In: *2017 International Conference on Engineering and Technology (ICET)*. Aug. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[23] Danny Roobaert and Marc M. Van Hulle. "A Natural Object Recognition System Using Self-Organizing Translation-Invariant Maps". In: *Neural Networks: Artificial Intelligence and Industrial Applications*. London, Great Britain: Springer-Verlag, 1995, pp. 151–154. ISBN: 9781447130871. DOI: 10.1007/978-1-4471-3087-1_29.

[24] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[25] Alex Krizhevsky, Ilya Sutskever, and Hinton Geoffrey. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[26] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. Beijing, China: O'Reilly, 2017. ISBN: 9781491914250.

[27] Carlo Innamorati, Tobias Ritschel, Tim Wayrich, and Niloy J. Mitra. "Learning on the Edge: Investigating Boundary Filters in CNNs". In: *International Journal of Computer Vision* 128.1 (Oct. 2019), pp. 773–782. DOI: 10.1007/s11263-019-01223-y.

[28] Anh-Duc Nguyen, Seonghwa Choi, Woojae Kim, Sewoong Ahn, Jinwoo Kim, and Sanghoom Lee. "Distribution Padding in Convolutional Neural Networks". In: *2019 IEEE International Conference on Image Processing (ICIP)* (Sept. 22, 2019–Sept. 25, 2019). Taipei, Taiwan: IEEE, 2019, pp. 4275–4279. ISBN: 9781538662496. DOI: 10.1109/ICIP.2019.8803537.

[29] Vivienne Sze, Yi-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[30] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, and Gang Wang. "Recent Advances in Convolutional Neural Networks". In: *Pattern Recognition* 77.C (May 2018). DOI: 10.1016/j.patcog.2017.10.013.

[31] Quiang Lan, Zelong Wang, Mei Wen, and Xhun-Yuan Zhang. "High Performance Implementation of 3d Convolutional Neural Networks on a GPU". In: *Computational Intelligence and Neuroscience* 2017 (Nov. 2017), pp. 1–8. DOI: 10.1155/2017/8348671.

[32] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. "Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function". In: *Neural Networks* 6.6 (June 1993), pp. 861–867. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5.

[33] Qilin Yin, Jinwei Want, Xiangyang Luo, Jiangtao Zhai, Sunil Kr. Jha, and Yun-Qing Shi. "Quaternion Convolutional Neural Network for Color Image Classification and Forensics". In: *IEEE Access* 7 (Feb. 2019), pp. 20293–20301. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2897000.

[34] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 21, 2017–July 26, 2017). Honolulu, HI: IEEE, 2017, pp. 1800–1807. ISBN: 9781538604571. DOI: 10.1109/CVPR.2017.195.

[35] Wnjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS ´16. Red Hook, NY: Curran Associates Inc., 2016, pp. 4905–4913. ISBN: 9781510838819.

[36] S. H. Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. "Impact of Fully Connected Layers on Performance of Convolutional Neural Networks for Image Classification". In: *Neurocomputing* 378 (Feb. 2020), pp. 112–119. DOI: 10.1016/j.neucom.2019.10.008.

[37] Michael Tschannen, Aran Khanna, and Anima Anandkumar. "StrassenNets: Deep Learning with a Multiplication Budget". In: Proceedings of Machine Learning Research 80 (July 2018). Ed. by Jennifer Dy and Andreas Krause, pp. 4985–4994.

[38] Roger Espansa and Mateo Valero. "Exploiting Instruction- and Data-Level Parallelism". In: *IEEE Micro* 17.5 (Sept. 1997), pp. 20–27. ISSN: 1937-4143. DOI: 10.1109/40.621210.

[39] Paul Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. 1st. Dallas, TX: The Pragmatic Bookshelf, 2014. ISBN: 9781937785659.

[40] David B. Kirk and Wen-Mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. San Francisco, CA: Morgan Kaufman Publishers Inc., 2013. ISBN: 9780124159921.

[41] Robert Love. *Linux Kernel Development*. 3rd. New York, NY: Addison-Wesley Professional, 2010. ISBN: 9780672329463.

[42] Xuanxia Yao, Peng Geng, and Xiaojiang Du. "A Task Scheduling Algorithm for Multi-Core Processors". In: *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings* (Sept. 2014), pp. 259–264. DOI: 10.1109/PDCAT.2013.47.

[43] Steve Carr, Jean Mayo, and Ching-Kuang Shene. "Race Conditions: A Case Study". In: *Journal of Comuting Sciences in Colleges* 17.1 (Oct. 2002), pp. 90–105.

[44] Karthikeyan Sankaralingam, S.W Keckler, W.R Mark, and D. Burger. "Universal Mechanisms for Data-Parallel Architectures". In: *Proceedings of the 36th Annual IEEE/ACM Interational Sumposium on Microarchitecture*. Washington, DC, Dec. 2003, pp. 303–314. ISBN: 076952043X.

[45] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 2326-3814. DOI: 10.1109/TC.1972.5009071.

[46] Jo Van Hoey. *Beginning x64 Assembly Programming: From Novice to AVX Professional*. 1st. New York, NY: Apress Media LLC., 2019. ISBN: 9781484250761.

[47] Daniel Kusswurm. *Modern x86 Assembly Language Programming: 32-Bit, 64-Bit, SSE, and AVX*. 1st. New York, NY: Apress, 2014. ISBN: 9781484200650. DOI: 10.5555/2723786.

[48] Sung-Jin Lee, Sang-Soo Park, and Ki Chung. "Efficient SIMD Implementation for Accelerating Convolutional Neural Network". In: *Proceedings of the 4th International Conference on Communication and Information Processing*. ICCIP ´18. New York, NY: Association for Computing Machinery, 2018, pp. 174–179. ISBN: 9781450365345. DOI: 10.1145/3290420.3290444. URL: https://doi.org/10.1145/3290420.3290444.

[49] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers's Manual*. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. Online; accessed 13 Mars 2020. 2016.

[50] Angela Pohl, Biagio Consenza, Mauricio Mesa, Chi Ching Chi, and Ben Juurlink. "An Evaluation of Current SIMD Programming Models for C++". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. WPMVP ´16. New York, NY: Association for Computing Machinery, 2016, pp. 1–8. ISBN: 9781450340601. DOI: 10.1145/2870650.2870653. URL: https://doi.org/10.1145/2870650.2870653.

[51] Vasileios Porpodas and Timothy M. Jones. "Throttling Automatic Vectorization: When Less is More". In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT ´15. Washington, DC: IEEE Computer Society, 2015, pp. 432–444. ISBN: 9781467395243. URL: https://doi.org/10.1109/PACT.2015.32.

[52] Intel Corp. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Online; accessed 10 February 2020. n.d.

[53] GNU Foundation. *6.52 Using Vector Instructions Through Build-in Functions*. https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html. Online; accessed 10 February 2020. n.d.

[54] The Rust Team. *SIMD for Faster Computing*. https://doc.rust-lang.org/edition-guide/rust-2018/simd-for-faster-computing.html. Online; accessed 10 February 2020. 2018.

[55] Ray Seyfarth. *Introduction to 64 bit Assembly Programming for Linux and OS X*. Scotts Valley, CA: CreateSpace Independent Publishing Platform, 2013. ISBN: 9781484921906.

[56] Nikola Zlatanov. *Computer Memory, Applications and Management [Internet]*. Santa Clara, CA, Feb. 2016. URL: https://www.researchgate.net/publication/295550090_Computer_Memory_Applications_and_Management.

[57] Mohammed Hassan. "On the Off-Chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?" In: *2018 IEEE Real-Time Systems Symposium (RTSS)* (Dec. 2018), pp. 495–505. ISSN: 1052-8725. DOI: 10.1109/RTSS.2018.00062.

[58] Milcho Prisagjanec and Pece Mitrevski. "Reducing Competitive Cache Misses in Modern Processr Architectures". In: *International Journal of Computer Science & Information Technology (IJCSIT)* 8.6 (Dec. 2016), pp. 49–57. DOI: 10.5121/ijcsit.2016.8605.

[59] AMD. *AMD Ryzen™ 7 3700x*. https://www.amd.com/en/products/cpu/amd-ryzen-7-3700x. Online; accessed 28 April 2020. 2019.

[60] Vivek Chaplot. "Cache Memory: An Analysis on Performace Issues". In: *International Journal of Advanced Research in Computer Science and Mannagement Studies* 4.7 (July 2016), pp. 26–28. ISSN: 2321-7782.

[61] Nathan Beckmann and Daniel Sanchez. "Modeling Cache Performance Beyond LRU". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Mar. 2016), pp. 225–236.

[62] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: *SIGPLAN Not.* 26.4 (Apr. 1991), pp. 63–74. ISSN: 0362-1340. DOI: 10.1145/106973.106981. URL: https://doi.org/10.1145/106973.106981.

[63] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. "Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*. New York, NY: Association for Computing Machinery, 2002. ISBN: 1581135424. DOI: 10.1145/774789.774804. URL: https://doi.org/10.1145/774789.774805.

[64] Syed Gilani, Nam Sung Kim, and Michael Schulte. "Scratchpad Memory Optimizations for Digital Signal Processing Applications". In: *211 Design, Automation Test in Europe* (Mar. 14, 2011–Mar. 18, 2011). Grenoble, France: IEEE, 2011, pp. 974–979. DOI: 10.1109/DATE.2011/5763158.

[65] Maryam Moazeni, Alex Bi, and Majid Sarrafzadeh. "A Memory Optimization Technique for Software-Managed ScratchPad Memory in GPUs". In: *2009 IEEE 7th Symposium on Application Specific Processors* (July 11, 2009–July 12, 2009). San Fransisco, CA: IEEE, 2009, pp. 43–49. DOI: 10.1109/SASP.2009.5226334.

[66] Alisdair Meredith. *Remove Deprecated Use of the register Keyword*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4340.html. Online; accessed 11 February 2020. 2014.

[67] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostic. "Make the Most out of Last Level Cache in Intel Processors". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. New York, NY: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303977. URL: https://doi.org/10.1145/3302424.3303977.

[68] Daniel Etiemble. "45-year CPU Evolution: One Law and Two Equations". In: *Second Workshop on Pioneering Processor Paradigms* (Feb. 2018).

[69] Luis F. Chaparro. "Chapter 0 - From the Ground Up!" In: *Signals and Systems using MATLAB*. Ed. by Luis F. Chaparro. Academic Press, 2011, pp. 3–62. ISBN: 978-0-12-374716-7. DOI: 10.1016/B978-0-12-374716-7.00002-8. URL: https://doi.org/10.1016/B978-0-12-374716-7.00002-8.

[70] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications". In: *IEEE Micro* 34.2 (Mar. 2014), pp. 34–43. ISSN: 0272-1732. DOI: 10.1109/MM.2014.12.

[71] Ruizhe Zhao, Wayne Luk, Xinyu Niu, Huifeng Shi, and Haitao Wang. "Hardware Acceleration for Machine Learning". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (July 3, 2017–July 5, 2017). Bochum, Germany: IEEE, 2017, pp. 645–650. ISBN: 9781509067626. DOI: 10.1109/ISVLSI.2017.127.

[72] Nicholas D. Lane and Petko Georgiev. "Can Deep Learning Revolutionize Mobile Sensing?" In: *Proceedings of the 16th Interational Workshop on Mobile Computing Systems and Applications*. HotMobile '15. New York, NY: Association for Computing Machinery, 2015, pp. 117–122. ISBN: 9781450333917. DOI: 10.1145/2699343.2699349.

[73] Zlatka Nikolova, Georgi Iliev, Miglen Ovtcharov, and Vladimir Poulkov. "Complex Digital Signal Processing in Telecommunications". In: *Applications of Digital Signal Processing*. Ed. by Christian Cuadrado-Laborde. Riejka, Croatia: IntechOpen, 2011. Chap. 1, pp. 3–23. ISBN: 978-953-307-406-1. DOI: 10.5772/26259. URL: https://doi.org/10.5772/26259.

[74] Mehdi R. Zargham. *Computer Architecture: Single and Parallel Systems*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996. ISBN: 9780130106612.

[75] Maria Elena Angoletta. "Digital Signal Processor Fundamentals and System Design". In: *CERN Accelerator School: Specialized Course on Digital Signal Processing* (May 31, 2007–June 9, 2007). Sigtuna, Sweden: CERN Publishing, 2007, pp. 167–229. ISBN: 9789290833116. DOI: 10.5170/CERN-2008-003.167.

[76] Chittoor V. Ramamoorthy and Hon F. Li. "Pipeline Architecture". In: *ACM Computing Surveys* 9.1 (Mar. 1977), pp. 61–102. ISSN: 0360-0300. DOI: 10.1145/356683.356687.

[77] Edward A. Lee and David G. Messerschmitt. "Pipeline Interleaved Programmable DSP's: Architecture". In: *IEEE Transactions on Acoustics, Speech and Signal Processing* 35.9 (Sept. 1987), pp. 1320–1333.

[78] Keshab K. Parhi and Debig G. Messerschmitt. "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding". In: *IEEE Transactions on Computers* 40.2 (Feb. 1991), pp. 178–195. ISSN: 0018-9340. DOI: 10.1109/12.73588.

[79] Preeti Rangan Panda, Namita Sharma, Srikanth Kurra, Khushboo Anil Bhartia, and Neeraj Kumar Singh. "Exploration of Loop Unroll Factors in High Level Synthesis". In: (Jan. 6, 2018–Jan. 10, 2018). Pune, India: IEEE, 2018, pp. 465–466. ISBN: 9781538636923. DOI: 10.1109/VLSID.2018.115.

[80] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558602860.

[81] Jack W. Davidson and Sanjay Jinturkar. *An Aggressive Approach to Loop Unrolling*. Tech. rep. USA, 2001.

[82] James Garland and David Gregg. "Low Complexity Multiply-Accumulate Units for Convolutional Neural Networks with Weight-Sharing". In: *ACM Transactions on Architecture and Code Optimization* 15.3 (Sept. 2018). ISSN: 1544-3566. DOI: 10.1145/3233300.

A Proofs of Correctness

This appendix proves the correctness of the developed algorithms formally.

A.1 Convolution

The convolutional layers are implemented using cross-correlation. While correct evaluation of a convolutional layer may require an arbitrary stride, the approach detailed to achieve this does not modify the evaluation of the layer. Instead, it simply discards the values that fall between the strides. As such, the correctness of the stride should be readily apparent under the assumption that the rest of the algorithm is correct. Similarly, the correctness of the approach used to emulate zero padding should be evident without further motivation.

It remains to prove the correctness of the approach used to compute the actual cross-correlation. The proof is constructed by first showing the correctness of the algorithm when computing one-dimensional cross-correlation. The result of this is used to prove the correctness of the algorithm when computing cross-correlation of an arbitrary dimension $d \in \mathbb{Z}^+, d > 1$.

A.1.1 Correctness of One-Dimensional Cross-Correlation

The correctness of the algorithm when computing one-dimensional cross-correlation is shown using mathematical induction. Assume the cross-correlation of an input $\mathbf{w} \in \mathbb{R}^{1\times m}$ and a filter kernel $\mathbf{k} \in \mathbb{R}^{1\times n}$ is to be computed. For the cross-correlation to be defined, it is required that $m \geqslant n$. The algorithm operates identically regardless of the position of the filter kernel and effectively computes the output of a FIR filter. As such, it can be concluded that if the application of the kernel at a certain position produces the correct result, so must also the application of the kernel at any other position.

One-dimensional, discrete cross-correlation of two functions f and g is given by

$$(f \star g)(j) = \sum_{i=1}^{n} f(i)g(j+i). \tag{A.1}$$

When computing the cross-correlation of two vectors, this may instead be expressed as

$$y_j = (\mathbf{u} \star \mathbf{v})_j = \sum_{i=1}^{n} u_i v_{j+i} \tag{A.2}$$

where the j^{th} component $y_j \in \mathbb{R}$ of the cross-correlation of the vectors $\mathbf{v}$ and $\mathbf{u}$ is computed. The notation $(\mathbf{v} \star \mathbf{u})_j$ is used to signify the j^{th} component of said cross-correlation. $v_i \in \mathbb{R}$ and $u_{j+i} \in \mathbb{R}$ are individual components of the vectors $\mathbf{v}$ and $\mathbf{u}$.

Step I

The base case is when $n \in \mathbb{Z}^+, n \leqslant 2$. When $n = 1$, the cross-correlation is computed as a single multiplication, the correctness of which is guaranteed.

To show the correctness of the algorithm when $n = 2$, let $\mathbf{w} \in \mathbb{R}^{1\times m}$ be defined as

$$\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \dots & w_{m-1} & w_m \end{bmatrix} \tag{A.3}$$

and $\mathbf{k} \in \mathbb{R}^{1\times 2}$ as

$$\mathbf{k} = \begin{bmatrix} k_1 & k_2 \end{bmatrix}. \tag{A.4}$$

The algorithm evaluates the cross-correlation of the first position in $\mathbf{w}$ as follows. The product of components w_1 and k_1 is stored in an accumulator $a \in \mathbb{R}$ as

$$a = w_1 k_1. \tag{A.5}$$

After this, the low lane of $\mathbf{w}$ is shifted out, yielding $\mathbf{w}'$ as given by

$$\mathbf{w}' = \begin{bmatrix} w_2 & w_3 & \dots & w_{m-1} & w_m & x \end{bmatrix} \tag{A.6}$$

for some $x \in \mathbb{R}$. Following the shift of the input, the contribution of the next term of the input is added to the accumulator a, producing $a' \in \mathbb{R}$ according to

$$a' = a + w_2 k_2 = w_1 k_1 + w_2 k_2. \tag{A.7}$$

By allowing $\mathbf{u} = \mathbf{k}$, $\mathbf{v} = \mathbf{w}$, $j = 0$ and $n = 2$, this is the exact result given by (A.2). As such, the algorithm produces the correct result also when $n = 2$.

Step II

Assume that the algorithm produces the correct result for a filter kernel of length $n = p$, $p \in \mathbb{Z}^+, p > 2$. For a filter kernel $\mathbf{k}$ of length $n = p + 1$, the cross-correlation shown in (A.2) may be rewritten as

$$y_j = (\mathbf{k} \star \mathbf{w})_j = \sum_{i=1}^{p+1} k_i w_{j+i} = \sum_{i=1}^{p} k_i w_{j+i} + k_{p+1} w_{j+p+1}. \tag{A.8}$$

The summation in the right-hand side computes the cross-correlation of a subset of p vector components, the correctness of which follows from the induction hypothesis. The addition of the product $k_{p+1} w_{j+p+1}$ is performed by shifting out the low value of the input, resulting in the low lane containing w_{j+p+1}. This value is multiplied by k_{p+1} and added to the final sum. That this yields the correct result follows from the same reasoning that was used to show the correctness for when $n = 2$ in Step I.

Step III

Step I shows that the algorithm produces the correct result for one-dimensional cross-correlation with kernels of length $n \in \mathbb{Z}^+, n \leqslant 2$. According to Step II, the values computed are correct also when $n = 2 + 1 = 3$, $n = 3 + 1 = 4$ and so on. As such, mathematical induction implies that the result is correct for kernels of all lengths $n \in \mathbb{Z}^+, n > 2$. This means that the algorithm must produce the correct result for one-dimensional cross-correlation of a signal with a kernel of arbitrary length $n \in \mathbb{Z}^+$.

A.1.2 Correctness of d-Dimensional Cross-Correlation

It remains to prove that the algorithm is correct when computing cross-correlation of dimensions 2 and 3. Instead of providing separate proofs for the two, this section proves the correctness for an arbitrary dimension $d \in \mathbb{Z}^+$. This is achieved by, again, using mathematical induction.

Step I

The base-case is one-dimensional cross-correlation, the correctness of which is proven in Section A.1.1.

Step II

The algorithm computes d-dimensional cross-correlation as a sum of a number of cross-correlations, each of dimension $d - 1$. These cross-correlations may be broken down recursively, eventually yielding a set of one-dimensional cross-correlations. Assume this approach produces the correct result for cross-correlation of an arbitrary dimension $d = p, p \in \mathbb{Z}^+, p > 1$. Then the correctness of the algorithm when computing cross-correlation of dimension $d = p + 1$ may be shown using contradiction.

Assume there exists a choice of input and filter kernel such that computing cross-correlation of dimension $p + 1$ as a sum of p-dimensional cross-correlations yields the wrong result. The formula for computing the cross-correlation of two functions f and g, both of dimension $p + 1$, is given by

$$(f \star g)(n_1, \ldots, n_p, n_{p+1}) = \sum_{i_1} \sum_{i_2} \cdots \sum_{i_p} \sum_{i_{p+1}} f(i_1, \ldots, i_p, i_{p+1}) g(n_1 + i_1, \ldots, n_p + i_p, n_{p+1} + i_{p+1}). \tag{A.9}$$

Let $h(i_1, n_1, \ldots, n_p, n_{p+1})$ denote the inner sums in (A.9), meaning

$$h(i_1, n_1, \ldots, n_p, n_{p+1}) = \sum_{i_2} \cdots \sum_{i_p} \sum_{i_{p+1}} f(i_1, \ldots, i_p, i_{p+1}) g(n_1 + i_1, \ldots, n_p + i_p, n_{p+1} + i_{p+1}). \tag{A.10}$$

As i_1 and n_1 are fixed in the right-hand side, it is evident that (A.10) computes a p-dimensional cross-correlation, the correctness of which follows from the induction hypothesis. Using (A.10), (A.9) may be rewritten as

$$(f \star g)(n_1, \ldots, n_p, n_{p+1}) = \sum_{i_1} h(i_1, n_1, \ldots, n_p, n_{p+1}) \tag{A.11}$$

Return now to the assumption of there existing at least one choice of input and filter kernel such that cross-correlation of dimension $p + 1$ may not be computed as a sum of p-dimensional cross-correlations. This runs counter to (A.11) which shows that cross-correlation of dimension $p + 1$ may indeed be computed as a sum of p-dimensional cross-correlations. Thus, a contradiction has been reached and hence, the algorithm must produce the correct result also when $d = p + 1$.

Step III

Step I shows that the algorithm computes cross-correlation correctly when $d = 1$. According to Step II, it yields the correct result also when $d = 1 + 1 = 2$, $d = 2 + 1 = 3$ and so on. Thus, mathematical induction implies that the algorithm computes the correct result for cross-correlation of an arbitrary dimension $d \in \mathbb{Z}^+$. ■

A.2 Max-Pooling

The algorithm evaluates the maximum of an $m \times n$ area of the image according to the principle

$$\max\left(\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}\right) = \max\left(\max\left(\begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix}\right), \max\left(\begin{bmatrix} a_{12} \\ \vdots \\ a_{m2} \end{bmatrix}\right), \dots, \max\left(\begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix}\right)\right). \tag{A.12}$$

The maximum value of each column is obtained by computing the column-wise maximums of row 1 and row i, $i = 2, 3, \dots, m$ over $m - 1$ steps. It is trivial to see that this produces a vector containing the column-wise maximum values of the input matrix. In order to prove the correctness of the entire algorithm, it remains to show that the method used to compute the component-wise maximum is correct.

Let $\mathbf{v}_c \in \mathbb{R}^{1\times n}$ denote the row-vector containing the column-wise maximum values of the input matrix as given by

$$\mathbf{v}_c = \left[\max\left(\begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix}\right), \max\left(\begin{bmatrix} a_{12} \\ \vdots \\ a_{m2} \end{bmatrix}\right), \dots, \max\left(\begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix}\right)\right]. \tag{A.13}$$

It follows that finding the global maximum of the $m \times n$ area under the kernel is equivalent to finding the maximum value in $\mathbf{v}_c$. This is done by copying $\mathbf{v}_c$, shifting out at most half the values under the kernel and computing the component-wise max of $\mathbf{v}_c$ and the shifted copy. Once this has been repeated up to $2\lfloor \log_2(n) \rfloor$ times, the maximum value of the $m \times n$ area under the kernel is held in the first component of the vector.

The correctness of the component-wise maximum is shown via mathematical induction. Let n denote the length of a vector $\mathbf{v}_c$ consisting of the maximum values of the columns under the kernel.

Step I

The base case of the induction is when $n \in \mathbb{Z}^+, n \leqslant 2$. If $n = 1$, the algorithm holds by definition.

To show the correctness when $n = 2$, let $\mathbf{v}_c \in \mathbb{R}^{1\times 2}$ be defined as

$$\mathbf{v}_c = \begin{bmatrix} v_{c_1} & v_{c_2} \end{bmatrix}. \tag{A.14}$$

When applying the algorithm, $\mathbf{v}_c$ is copied to another vector $\mathbf{v}'_c \in \mathbb{R}^{1\times 2}$, the low lane of which is shifted out. This means that $\mathbf{v}'_c$ is given by

$$\mathbf{v}'_c = \begin{bmatrix} v_{c_2} & x \end{bmatrix} \tag{A.15}$$

for some $x \in \mathbb{R}$. The component-wise maximum of $\mathbf{v}_c$ and $\mathbf{v}'_c$ is computed and stored in $\mathbf{v}_c$. Once this is done, the first component of $\mathbf{v}_c$ contains the larger of v_{c_1} and v_{c_2}. Hence, the algorithm produces the correct result also when $n = 2$.

Step II

Assume the algorithm produces the correct result for a vector of arbitrary length $n = p, p \in \mathbb{Z}^+, p > 2$. For a vector $\mathbf{v}_c$ of length $n = p + 1$, it must hold that its maximum value is either the maximum of the first p components or the value of component $p + 1$. That the maximum value computed for the first p components is indeed the maximum of these components follows from the induction hypothesis.

That component $p + 1$ is considered correctly is shown as follows. Let v_{c_i} denote the value originally in component $i \in \mathbb{Z}^+, i \leqslant p + 1$ of $\mathbf{v}_c$. If $v_{c_{p+1}}$ is the maximum value of $\mathbf{v}_c$, then after having computed the maximum of the first p components, the second component of $\mathbf{v}_c$ contains $v_{c_{p+1}}$. This follows from the fact that for a vector of length $n \in \mathbb{Z}^+, n > p$, its components $2, 3, \ldots, p, p + 1$ may be treated as a separate vector of length $n = p$. As such, the induction hypothesis gives that the value in the second component of $\mathbf{v}_c$ must be the maximum of components $2, 3, \ldots, p, p + 1$.

Since, at this point, the first component of $\mathbf{v}_c$ contains the maximum of $v_{c_i}, i = 1, 2, \ldots, p$ and the second component the maximum of $v_{c_j}, j = 2, 3, \ldots, p, p + 1$, the greater of the two must be the maximum value of $\mathbf{v}_c$. As shown in Step I, copying the vector, shifting out its lowest component and performing a component-wise maximum computes the maximum of the first two components. As such, the algorithm produces the correct result when $v_{c_{p+1}}$ is the global maximum of $\mathbf{v}_c$.

It remains to consider the case when $v_{c_{p+1}}$ is not the global maximum of $\mathbf{v}_c$. There are 2 potential cases. If $v_{c_{p+1}}$ is the maximum value of components $2, 3, \ldots, p, p + 1$, then the induction hypothesis gives that component 2 of $\mathbf{v}_c$ contains $v_{c_{p+1}}$ for the final step. During this step, $v_{c_{p+1}}$ is compared to v_{c_1} and found to be lesser. As such, the algorithm yields the correct result in this case.

If $v_{c_{p+1}}$ is not the maximum value of components $2, 3, \ldots, p, p + 1$, then component 2 of $\mathbf{v}_c$ contains one of the values $v_{c_i}, i = 2, 3, \ldots, p$ for the final step. The correctness of this case follows directly from the induction hypothesis.

Step III

Step I shows that the algorithm produces the correct result when the vector is of length $n \in \mathbb{Z}^+, n \leqslant 2$. According to Step II, it yields the correct result also when $n = 2 + 1 = 3$, $n = 3 + 1 = 4$ and so on. As such, mathematical induction implies that the assumption of correctness holds for all $n \in \mathbb{Z}^+, n > 2$. This, in turn, implies that the algorithm produces the correct result for kernels of any width $n \in \mathbb{Z}^+$. ■

A.3 Fully Connected

The proof is by contradiction. Assume there exists at least one choice of weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, input vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$ and bias vector $\mathbf{b} \in \mathbb{R}^{m \times 1}$ such that the algorithm used to evaluate the fully connected layer produces the wrong result. Let $\mathbf{y} \in \mathbb{R}^{m \times 1}$ be the result vector. Using this nomenclature, the evaluation of a fully connected layer can be expressed as

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{A.16}$$

Let $\mathbf{y}' \in \mathbb{R}^{m \times 1}$ be the vector resulting from the matrix-vector multiplication according to

$$\mathbf{y}' = \mathbf{W}\mathbf{x}. \tag{A.17}$$

This allows for expressing a fully connected layer as

$$\mathbf{y} = \mathbf{y}' + \mathbf{b}. \tag{A.18}$$

While the algorithm effectively computes the vector addition of several 16-component vectors and concatenates them to produce the output, this is trivially equivalent to the addition in

(A.18). As this addition holds by definition, it remains only to prove the correctness of the implementation of the matrix-vector multiplication.

Evaluating a component y'_i, $i = 1, 2, \ldots, m$ of the vector resulting from the matrix-vector multiplication is equivalent to computing the dot product of $\mathbf{x}$ and the transpose of row i of the matrix. More formally, this is expressed as

$$y'_i = \mathbf{W}^T_{i*} \cdot \mathbf{x} \tag{A.19}$$

where the notation $\mathbf{W}_{i*} \in \mathbb{R}^{1\times n}$ is used to signify the entirety of row i in $\mathbf{W}$. The dot product may be rewritten as the sum of the component-wise product of $\mathbf{W}^T_{i*}$ and $\mathbf{x}$ according to

$$\mathbf{W}^T_{i*} \cdot \mathbf{x} = \sum_{j=1}^{n} W^T_{ij} x_j. \tag{A.20}$$

The sum of component-wise products may, without loss of generality, be broken up into a sum of sums as

$$\sum_{j=1}^{n} W^T_{ij} x_j = \sum_{j=1}^{32} W^T_{ij} x_j + \sum_{j=33}^{64} W^T_{ij} x_j + \ldots + \sum_{j=n-\lfloor \frac{n}{32} \rfloor}^{n} W^T_{ij} x_j. \tag{A.21}$$

The right-hand side of (A.21) may be expressed on the more concise form

$$\sum_{k=0}^{\lfloor \frac{n}{32} \rfloor - 1} \sum_{j=32k+1}^{32(k+1)} W^T_{ij} x_j + \sum_{j=n-\lfloor \frac{n}{32} \rfloor}^{n} W^T_{ij} x_j \tag{A.22}$$

where the separate sum on the far right is required to correctly handle the case when n is not a multiple of 32. The order of the nested summations in (A.22) may be interchanged, yielding

$$\sum_{j=0}^{31} \sum_{k=j\lfloor \frac{n}{32} \rfloor + 1}^{(j+1)\lfloor \frac{n}{32} \rfloor} W^T_{ik} x_k + \sum_{j=n-\lfloor \frac{n}{32} \rfloor}^{n} W^T_{ij} x_j. \tag{A.23}$$

While the algorithm intermixes the summations of up to 16 rows, each row is still computed using the approach outlined in (A.23).

Return now to the assumption of there existing at least one set of inputs $\mathbf{W}$, $\mathbf{x}$ and $\mathbf{b}$ such that the algorithm produces the wrong result. This implies that these choices of input must not satisfy at least one of the equivalences leading up to (A.23). As the equivalences hold for any and all choices of input, this is a contradiction and thus, the algorithm must be correct. ■