

UNIVERSITÄT
DES
SAARLANDES

FACHRICHTUNG 6.2
INFORMATIK

Integrating a Denoising Neural Network Into a Renderer Using a High-Level Functional Language and Partial Evaluation

Bachelorarbeit im Fach Informatik

Name:
Joshua Meyer

Matrikelnummer:
2573323

Erstgutachter:
Prof. Dr. Philipp Slusallek

Zweitgutachter:
Prof. Dr. Sebastian Hack

15. Dezember 2021

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

_____ _____

Ort, Datum *Unterschrift*

Abstract

Convolutional Neural Networks (CNN) are used widely in computer vision and image processing tasks. Recent technological progress even allows real-time usage for some tasks. Removing Monte-Carlo noise created by ray tracing is one of them. In this thesis, we present the integration of a denoising neural network into a ray tracer using the compiler framework AnyDSL to allow real-time denoising. We use the high-level, functional language Artic to implement the forwarding propagation of a general CNN. To compare our implementation's performance with state-of-the-art solutions, we provide benchmarks that we ran with our implementation and implementations utilizing widely used pre-existing libraries like Intel's oneAPI. On the CPU, we achieved a runtime that is comparable and even surpasses the performance of oneAPI. The integration only serves as a proof-of-concept example. The code presented in this work can, in the end, be used for arbitrary CNNs.

Table of Contents

1 Introduction . . . 1

2 Background and Related Work . . . 3
2.1 AnyDSL . . . 3
2.2 Rodent . . . 4
2.3 Denoising Libraries . . . 4
2.4 Machine Learning Libraries and Frameworks . . . 5
2.5 Linear Algebra Libraries and Similar . . . 6

3 Neural Networks . . . 7
3.1 Artificial Neural Networks . . . 7
3.2 Convolutional Neural Networks . . . 8
3.2.1 The Convolutional Layer . . . 8
3.2.2 Common Layers in CNNs . . . 11
3.3 Implementing Feedforwarding in AnyDSL . . . 13
3.3.1 Preliminary Functionality . . . 13
3.3.2 The Matrix Multiplication . . . 15
3.3.3 The im2col Rearrangement . . . 18
3.3.4 The Convolution Function . . . 21
3.3.5 Pooling and Upsampling Layers . . . 22
3.3.6 Other Backends . . . 23

4 Ray Tracing and Denoising . . . 27
4.1 Ray Tracing . . . 27
4.2 Integration Into Rodent . . . 28

5 Results . . . 31
5.1 Real-Time Viability . . . 33

6 Conclusion and Future Work . . . 35

1. Introduction

Artificial neural networks (ANN) have been an interesting theoretical concept for a long time already. Motivated by the way the brain works, an ANN consists of numerous interconnected neurons to convey information from one end to the other. During learning, every neuron is assigned a weight, which corresponds to its importance throughout the computation. In that way, an ANN can be interpreted as a parameterized system. A neural network with many neurons can learn and memorize more information. However, with a growing number of neurons, it is also much harder to train them accordingly. Not only does one need an enormous amount of training data, but it is also computationally expensive, making it a time- and resource-intensive process. With every passing year, newly developed hardware makes computers faster and more powerful. With the internet and all the data it contains and can be shared with it, training data is more readily available for many tasks.

Thus, practically researching more complex and larger neural networks is made possible. Early works already show the capabilities of ANNs. While most pioneering studies used multilayer perceptron (MLP) based neural networks even for image processing tasks [30, 6], more recent work on image tasks relies on convolutional neural networks (CNN). In contrast to MLPs that operate globally by connecting all neurons, CNNs provide a local approach and reduce the number of learned parameters in a neural network by a large factor. They also remove (or loosen) the input size restriction of MLP networks, as is explained in chapter 3.

In 2010, Krizhevsky, Sutskever, and Hinton trained a deep CNN to classify images into 1000 different classes in the context of the ImageNet LSVRC-2010 contest with huge success. Two years later, in the same contest, they improved their results further with another variant of their model. Among others, their results, presented in [26], lead to CNNs instead of MLPs becoming the standard for image tasks.

While neural networks gained attention and popularity over the past and training them does no longer pose as big a problem, their usage in real-time applications is still limited. In order to make them applicable in real-time, their evaluation needs to be extremely fast, while the computer also has to perform the computations of the actual task. This is particularly hard to achieve for CNNs with large images as input during rendering, for example.

Ray tracing, an old concept as well, is also becoming applicable in real-time applications and games with very recent hardware. However, as ray tracing uses Monte-Carlo methods as a means to solve an integral, called the rendering equation, it is also prone to Monte-Carlo noise. An approach to use ray tracing to render stunning images regardless is to pair it with an appropriate denoising technique. This is where convolutional neural networks come into play. Traditionally, denoising used deterministic filtering techniques and mathematical analysis to restore noise-free images (e.g., [11]). In 2005, Buades, Coll, and Morel review many such methods and even present a new one in the same paper [5]. With the abundance of data and the power of CNNs, training and using CNNs for denoising is a real possibility. Nvidia, with Optix [7], and Intel, with Open Image Denoise [20], show that using CNNs for this task is more than suitable and often yield better results than the traditional methods. The remaining problem is the real-time viability since the computational complexity of CNNs exceeds the one of filtering techniques by far.

In this thesis, we present a way to efficiently integrate a denoising convolutional neural network in a ray tracer using a functional language and test its real-time viability. To this end, we first take a look at related work and their approaches to achieve similar goals in chapter 2. At the same time, we explain the libraries and programs we need throughout the rest of the thesis. In chapter 3, we introduce the concept of neural networks in more detail. We start with MLPs and explain how they turn input into output values. These results are then used to understand the way CNNs work. The crucial point here is the convolutional layer, to which CNN's owe their name. However, there are also other commonly used layers of importance in CNNs. Having understood CNNs, we explain our implementation of a forward propagation in AnyDSL in detail by justifying the design choices and providing code snippets. In chapter 4, we use the resulting code from chapter 3 to integrate a denoising neural network in Rodent, a ray-tracing library written in AnyDSL. This integration is used as a proof-of-concept to show the functionality of our implementation and to test its real-time viability. In chapter 5, we present some benchmarks comparing the efficiency and performance of our implementation with implementations that rely on external libraries to perform the convolutions. With a brief summary, chapter 6 concludes this paper while presenting thoughts on future work which could extend the results of this thesis.

2. Background and Related Work

In this chapter, we review related work in the area of denoising and fast neural network evaluation. We start by introducing AnyDSL, a compiler framework used throughout this paper for all our implementations. Then, we shortly present a Monte-Carlo ray tracer written in this framework in which we integrate our denoiser. To have a comparison baseline for our results and performance, we briefly talk about select denoising and machine learning libraries. Furthermore, as matrix operations are crucial in neural network computations, we take a look at linear algebra libraries, which are, among other use cases, used to speed up forward and backward propagation of neural networks.

2.1 AnyDSL

High-performance code is often written in languages like C/C++ as they provide access to very low-level instructions and are, therefore, well suited for tuning performance. However, common C compilers, like, e.g., clang [9], do not support some features that could further speed up the generated program. For example, they do not use *partial evaluation* to specify functions at compile time to known arguments and do not inherently turn functions into *continuation-passing style* (CPS), which would allow programs to continue their computations without returning, even though it might be possible.

AnyDSL is a partial evaluation compiler framework [32] to write high-performance code. The framework supports two languages: Impala and Artic. Artic is an acronym for "AlteRnaTive Impala Compiler" and serves as an extension to Impala while also introducing some syntax changes, leaving Impala mostly deprecated.

Essentially, AnyDSL's frontend is only syntactic sugar for a higher-order, functional intermediate representation called Thorin [31]. Artic is translated to Thorin, which then applies further optimizations and compiles to CUDA [37], OpenCL [47], and LLVM [28]. The resulting LLVM code is analyzed with the vectorization tool RV presented in [35] and then finally compiled to the underlying hardware architecture.

A notable difference compared to C is that, as Thorin is a functional intermediate representation, Artic is a functional language. As a component of AnyDSL, it allows partially evaluating functions by annotating them with @. Additionally, higher-order functions are natively supported, and their usage is highly encouraged. In many cases, using these higher-order functions helps the partial evaluator to produce better performant code. Thorin, as the underlying intermediate representation, uses continuation-passing style. To keep implementations simple, the programmer is not forced to write CPS functions since the code is compiled to CPS automatically. Moreover, AnyDSL and C are directly connected. It is possible to call functions from AnyDSL in C, and vice versa, by exporting or importing them, respectively. The compiler frontend clang [9] can then be used to compile and link the code to create an executable. Sharing data needed in both codebases is possible by passing pointers, and the data does not need to be copied. Copying every time data is changed or used in the other language would be a huge bottleneck and go against the spirit of creating high-performance code. Allowing the sharing of pointers tackles this problem elegantly. However, the correctness of the interfaces and typing is the programmer's responsibility, as the framework does not check it.

AnyDSL, Artic, is the programming language of choice throughout the rest of this thesis for all our implementations. If we want to call specific libraries or API's, this is easily possible by creating an interface to C and calling that API function from C.

2.2 Rodent

Since we want to integrate a denoising neural network in a renderer, we need to decide on a particular one. We could write a renderer ourselves or use an open-source one for which the integration would not be too much work. As we decided to program in the AnyDSL compiler framework, a renderer using the same framework would be the best fit for this, even though choosing an arbitrary renderer written in C would work just as well.

In 2019, Pérard-Gayot et al. present a rendering library, Rodent [44], which uses the AnyDSL compiler framework. Using AnyDSL's partial evaluation feature, Rodent generates renderers specialized to a specific scenery. Consequently, they are fast and high performant once compiled, but the compilation itself can take some time. In Rodent, Monte-Carlo ray tracing is used as a means to approximate the real result of the rendering equation [24]. Thus, Monte-Carlo noise is inherent to all generated renderers. Filtering the noise often is done by increasing the number of samples per pixel. This not only significantly increases rendering times but also becomes less effective the more samples have been taken already. For artifacts arising from numerical errors, it may not even be possible to remove them by increasing the sample count. Special filtering techniques can be used to blur noisy, ray-traced images and remove said noise [5, 11]. Providing these filters guidance images, like normal, depth, and albedo buffers, helps them be content- and edge-aware and preserve detail with only selective blur. In [16], the authors, He, Sun, and Tang, discuss guided filters and compare the accuracy and quality of the filtered images with images that were filtered using unguided techniques. The guided filters are, however, still prone to losing high frequencies and sharp edges. Recently, machine learning is becoming attractive to be used for denoising rendered images, for example, with a denoising autoencoder [7, 20]. While this approach seems very promising and denoises with great success, it is computationally much more expensive than simple guided filters.

2.3 Denoising Libraries

Denoising is not a necessary step in ray tracing, but today's Monte Carlo ray tracing methods are unable to get rid of all the noise in an image in a reasonable amount of time. As mentioned before, denoising with neural networks is a new and promising approach. There are already libraries with pre-trained neural networks available one can try and experiment with. However, "noise" is a vague term as many different kinds of noise exist. They look distinct, emerge differently and also need other ways of handling [43]. For example, Guo et al. present an approach of denoising real photographs, whose noise strongly differs from the noise created by renderers [15]. Therefore, a neural network needs specific training according to the noise it supposedly should filter. From here on, whenever we refer to the term "noise", we are talking about Gaussian noise, arising from the Monte-Carlo methods.

In recent years, Nvidia has been pushing real-time ray tracing, and with it, its neural networks, to enhance image quality. Especially for real-time ray tracing, generating stunning, noise-free images while retaining a low sample count is essential as rendering more samples is expensive. In their ray tracing engine, Optix [7], they are using a denoising autoencoder as a post-processing step.

There is also Intel Open Image Denoise (OIDN) [20], which is a free-to-use, open-source, high-performance denoising library meant to be used on ray-traced images. It is based on a

convolutional neural network and provides a pre-trained model. This library comes with an easily accessible C/C++ API to integrate OIDN into a ray tracer. Therefore, OIDN is a good starting point to integrate into Rodent. The autoencoder takes as input three RGB images: The noisy rendered image, the normal image, and the albedo image. Combined, this makes nine input channels, which are processed, with convolutional, pooling, and upsampling layers to create three output channels. In section 3.2 we learn how these layers operate. Intel accelerates the computation using their linear algebra library oneAPI [19]. With this library, they are able to exploit modern hardware architectures and achieve high performance on them.

2.4 Machine Learning Libraries and Frameworks

There are many machine learning libraries and frameworks provided by different companies. Early examples for such libraries would be Torch7 [10] or Caffe [22]. Torch7 is mainly written in the script language Lua [18] and C. Torch7 relies on BLAS and LAPACK libraries to speed up its computations and utilize SIMD instructions, which it does not do in its code otherwise. To parallelize the workflow, OpenMP [12] is used, which simplifies the parallelization to only an annotation. Caffe was written mainly in C++ and CUDA, ensuring that low-level and efficient code is used in their backend. It provides bindings to Python and Matlab to have an easily accessible API outside of C++.

However, more recently, frameworks like TensorFlow [1], and PyTorch [42] are favored and primarily used nowadays. TensorFlow's first version was released in 2015 by Google. The languages used are C++, CUDA, and Python. It is known to be usable in a wide variety of programming languages and can, therefore, be integrated into many applications. In the beginning, TensorFlow was focused heavily on multi-core CPU setups, but as it became apparent that machine learning could take advantage of the parallel computing capability of GPU's, GPU support was added.

PyTorch is a high-performance, machine learning Python library based on the beforementioned Torch library. Its goal is to be flexible and have good performance while still being easy to use. As it is the most recent library among the mentioned ones, by the time it was developed, it was already known that GPU support would be necessary for efficient implementations. Thus, it inherently comes with GPU hardware acceleration. By providing syntax and structure similar to other already known python libraries, like NumPy [40] or Pandas [34], the framework is easy to use. All listed libraries are open-source, as Python libraries usually are.

For our work, the importance of the machine learning library used lies in the ability to train and export models since we are only interested in a high-performance forwarding solution. As PyTorch provides us with precisely that and is the most recent library here, it is our library of choice. We will use it to train a neural network, export its learned weights and import them into our codebase. However, this choice is purely by preference. As long as the machine learning library supports training and exporting the weights, it could be used here instead of PyTorch.

With the ability of the libraries to propagate forward through the network, we can also test our implementation for errors by comparing the output of networks in PyTorch with the corresponding outputs of the forwarding in AnyDSL.

2.5 Linear Algebra Libraries and Similar

Using efficient matrix-matrix operations is a significant part when propagating a neural network. More often than not, when using matrices for calculations, it makes sense to fall back to known efficient implementations other people have made available already. Basic Linear Algebra Subroutines (BLAS) libraries contain subprograms to efficiently calculate matrix operations, even taking advantage of modern hardware. For example, they utilize SIMD instructions to perform 4 additions with one instruction instead of 4. Although highly optimized, such libraries can still be less optimal than a specialized implementation of select subroutines. This is because a library needs to be flexible and applicable to many different problems. However, for one specific issue, one might know more about the structure of the matrices or operations that need to be done on top of the ones the library offers. This knowledge can be combined with the subroutine's algorithm when not using a library, making it more efficient.

There are many BLAS libraries such as cuBLAS [3], BLIS [49], oneAPI [19], OpenBLAS [41] and many more. Their main difference often is the intended underlying hardware and implementation language. For example, cuBLAS is intended for use on Nvidia GPU's only, since it uses CUDA [37]; BLIS is an open-source CPU library from AMD; oneAPI for use on CPU and Intel GPUs. The language subprograms usually are implemented in is either C or Fortran. Both provide access to very low-level instructions and are, therefore, well suited for tuning performance.

In this work, we use oneAPI as an example for a BLAS library on the CPU since it is the library Intel uses for its denoising library OIDN. It serves as a reference to compare our implementations. Since Nvidia's deep neural network library cuDNN [38] uses cuBLAS to optimize its computations, cuBLAS will be our reference on the GPU side.

3. Neural Networks

We start this chapter by introducing artificial neural networks and explaining how they work. In particular, we take a closer look at convolutional neural networks. This subset of neural networks uses convolutions to propagate the input and generate an output. Finally, we discuss the implementation of convolutional neural networks and implement an efficient feed-forward routine in AnyDSL.

3.1 Artificial Neural Networks

In machine learning, an artificial neural network is a network of connected neurons. These connections have a weight that is multiplied with the source neurons' value when contributing to the destination neuron. The neurons themselves are often grouped into different layers. There is an input layer, some hidden layers, and last but not least, the output layer. The layers can consist of variant numbers of neurons. Altogether, the counts sum up to the total number of neurons in the network.

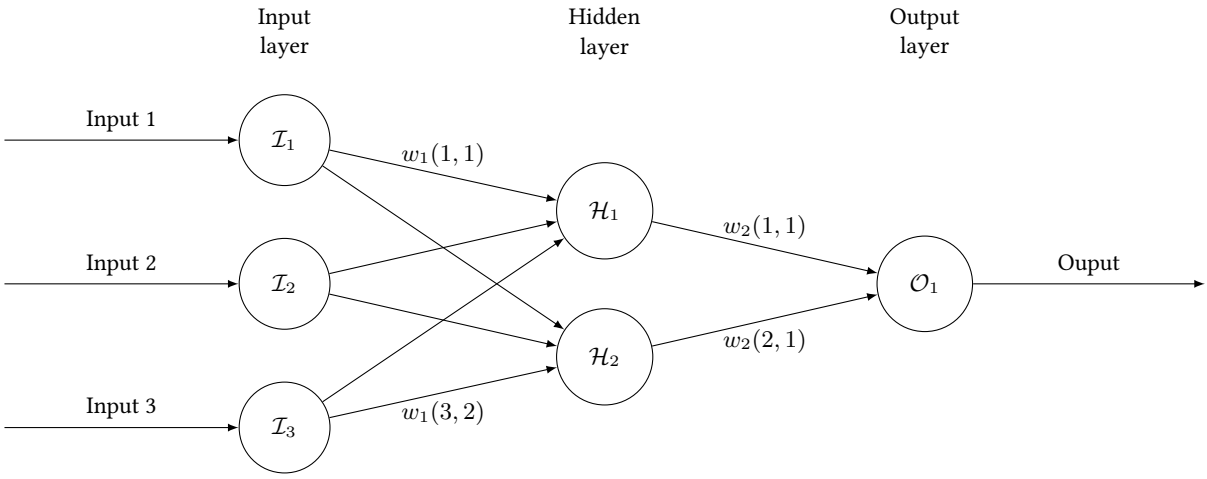


Fig. 1: A simple fully-connected neural network with 3 inputs, 1 hidden layer with 2 neurons, and an output layer with 1 output

In Figure 1 for example, we have a neural network with three input neurons, one hidden layer with two neurons, and one output neuron in the output layer. The neurons in a layer affect all neurons from the next layer. Therefore, this network is called a *fully-connected neural network* or a *multilayer perceptron (MLP)*. Theoretically, it is possible to have connections between neurons in the same layer or even to previous layers. Networks having this property are then called *recurrent* and are not the subject of this work. We restrict ourselves to *feed-forward* neural networks in which every neuron only has connections to neurons in the following layers. When the feed-forward network in Figure 1 is fed an input consisting of three values, the calculation for the hidden neurons is given by

$$\mathcal{H}_1 = f\left(\sum_{i=1}^{3} w_1(i,1)\,\mathcal{I}_i + b_1\right), \qquad \mathcal{H}_2 = f\left(\sum_{i=1}^{3} w_1(i,2)\,\mathcal{I}_i + b_2\right),$$

where $\mathcal{H}_i$ is the i-th neuron in the hidden layer, $\mathcal{I}_i$ the i-th neuron in the input layer, $w_1(i,j)$ the weight of the connection from $\mathcal{I}_i$ to $\mathcal{H}_j$, b_i a constant bias for $\mathcal{H}_i$ and $f : \mathbb{R} \to \mathbb{R}$ a function. The same calculation can be repeated to receive the output value from the hidden layers' values:

$$\mathcal{O}_1 = f\left(\sum_{i=1}^{2} w_2\left(i,j\right)\mathcal{H}_i + b\right).$$

The function f used in the above calculations is called an *activation function*. An activation function is a function $f : \mathbb{R} \to \mathbb{R}$ applied to each neuron after its value calculation to activate it. Commonly used activation functions include but are not limited to the sigmoid function

$$\varphi^{\text{sig}}(t) = \frac{1}{1+e^{-t}},$$

and rectifier (rectified linear units, also ReLU), e.g.

$$\varphi^{\text{relu}}(t) = \max(0,t).$$

Every activation function has different up- and downsides. For example, the sigmoid function is popular for its differentiability. Nwankpa et al. provide a thorough comparison and overview over commonly used activation functions in [39]. As ReLUs are becoming more and more popular recently, Xu et al. also provide a comparison between different types of ReLUs [50]. Overall, activation functions are essential for neural networks since they add nonlinearity to the computation path. Without them, one could express many neural networks as a simple matrix multiplication, and the concept of layers would not add complexity.

The most significant learned parameters are the weights and biases. Almost everything else is either a static design choice or a *hyperparameter*. Hyperparameters are parameters that are fixed before learning even begins. Choosing the best design for a given task is not easy and the subject of ongoing research. Liu et al. survey different architectural choices and their motivations in [33] to provide a guideline. Then, there are other paper's whose main contribution is the investigation and comparison of different network sizes [17]. For example, one can be sure that a network with many neurons is capable of learning what it is supposed to. But giving a network too many neurons means that it easily overfits its training data and hurts generalization performance. Having too few neurons may leave the network with not enough flexibility to adapt and learn the patterns of the dataset.

Once one has decided on architecture and size, one needs to train the network with existing data. Training is done by propagating training data through the network and evaluating the result based on a *loss function* [21, 53]. With a backward pass, the neurons' weights are tuned based on a learning rate and their contribution to the wrong result. However, training neural networks is a whole research area of its own. Because of their complexity, the training process is hard to understand and, often, trial and error [14]. Diving deeper into the training problematic is beyond the scope of this thesis.

3.2 Convolutional Neural Networks

3.2.1 The Convolutional Layer

Convolutional neural networks (CNN) are a subset of artificial neural networks that use convolutions during their forward propagation. In a *convolutional layer*, there is a *kernel* containing learned weights. This kernel defines a receptive field that slides over the whole input

space. At each step, the element-wise product of the weights with the current receptive field of the input is calculated. Summing up all values in the resulting matrix generates a single output value. Grouping all the output values together yields a new layer in the path through the neural network.

One can perform convolutions on inputs with arbitrary dimensions. Figure 2 shows one step of a 2D-convolution and the formula of the calculation. On the left, we see a two-dimensional input volume. For a convolution, we take a receptive field, compute its element-wise product with the 2D kernel and sum up the entries in the results, which then yields one output value for the output volume. As we can see, while the input volume is of size 8×8, the output volume only has size 6×6. This is because we only consider 3×3 receptive fields over the input, and we can only fit 36, not 64, of them on an 8×8 grid. To achieve the same number of output neurons as input ones, we need to add a *padding* to the input volume. A padding transforms the input by adding values to the borders, increasing the input's size. In this particular case, enlarging the 8×8 grid to a 10×10 grid would result in the output having the same size as the input. In general, the padding size to achieve this depends on the kernel size. If we want to keep the input's size, we need to fit the correct amount of receptive fields onto it. In every dimension, we need a padding P of

$$P = \frac{K-1}{2},$$

where K is the kernel size in the current dimension. One can freely choose the values to put into the padding. Often they are just constant 0, but they can also be correlated to the input, e.g., by using the nearest-neighbor value. There is one more hyperparameter one can change to achieve different results. Namely, the *stride*. The stride S defines how far the kernel moves over the input after each step. E.g., in the explanation above, we used a stride of 1. With these hyperparameters, we also get the constraint of

$$\frac{W-K+2P}{S}+1 \in \mathbb{N}$$

to have a valid configuration, where W is the input size (in the respective dimension). An invalid configuration would mean that the kernel cannot move symmetrically across the input. If we look at Figure 2, a stride of 2 would be invalid since the last column of the input would still be a center of the kernel, but the kernel does not fit there anymore. Our constraint also captures this since

$$\frac{8-3+2\cdot 0}{2}+1 = \frac{5}{2}+1 = 3.5 \notin \mathbb{N}\,.$$

While in MLPs, every output neuron is affected by every input, we can observe that in a convolutional layer, each output neuron only has connections to a small subset of input neurons. This local connectivity instead of a global one lets us deal with high-dimensional inputs, such as images, with far less effort. Having a fully-connected layer for a 200×200 image, we get $200 \cdot 200 \cdot 3 = 120000$ weights the network needs to learn. With a convolutional layer with kernel size 3×3, there are only $K \cdot K \cdot 3 = 27$ weights that are shared over the whole image (Note the extra factor of 3 in both cases for the RGB channels.). By these calculations, one can easily see what happens to the number of weights when the image is scaled up by a constant factor c. While the number of neurons is multiplied by c^2 in the MLP case, it stays the same in the convolutional case. This results in the fully-connected layer depending on the input size. If we wanted to input images of different sizes to our network, we would need several fully-connected networks; one for each size. A CNN does not have this restriction. It simply

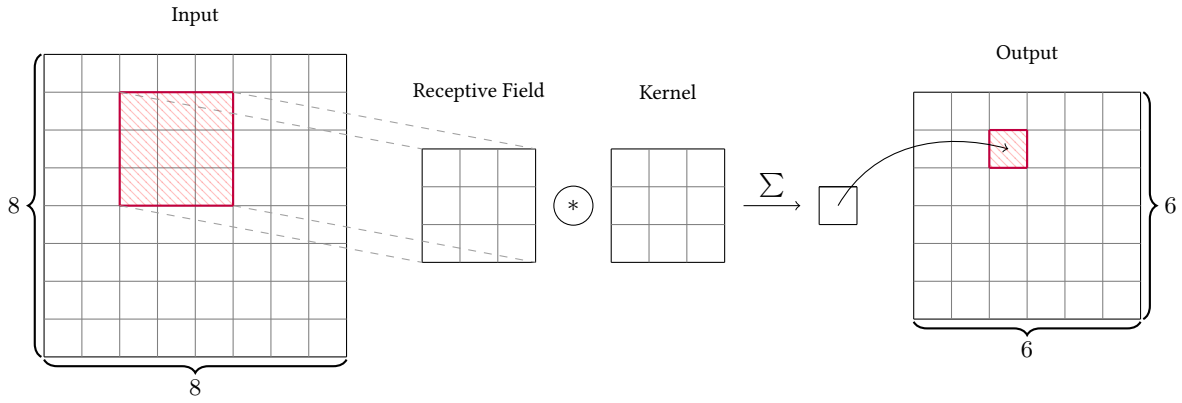


Fig. 2: Depiction of a sample 2D convolution without padding. The output value is $f\left(\sum_{i=1}^{3}\sum_{j=1}^{3} w(i,j)\mathcal{R}_{i,j} + b\right)$, where $\mathcal{R}_{i,j}$ is the value in row i and column j in the receptive field and $w(i,j)$ the weight in the kernel at position (i,j). Like in the MLP case b is a constant bias and f is an activation function.

has to run the kernel over more input values than before but can use the same network to do so.

Another side effect of the local connectivity and the shared weights in the learned kernel is *feature learning*. In contrast to a non-convolutional neural network, a CNN can learn features across the image space, independent of their position. As such, a CNN can, for example, memorize what a face looks like, with one or several kernels once, and then apply it to the whole image space. A fully-connected neural network would need to learn the features at every position from anew. Partly for these two reasons, convolutional neural networks became very popular in image processing and computer vision tasks. But CNNs also have their downsides. For example, their computational complexity is relatively high. While we can use a matrix multiplication to evaluate a fully-connected layer, we cannot do this for a convolutional layer without additional effort. As we discuss in section 3.3, there are ways to rearrange the input and kernel to use matrix multiplication, but it is not as trivial and efficient as it is for a fully-connected layer.

Having introduced 2D convolutions, we can now look at 3D convolutions. In the calculation for the number of weights above, we already hinted at the third dimension of convolutions when using them for images. Every pixel of an image has three values defining it: Red, green and blue. When processing an image, we need to take all these *channels* into account. Therefore, our input, as well as our kernel, have to be three-dimensional. Channels are also called *feature maps*, and we are not limited to only having three of them. In fact, CNNs often take images as input and, thus, start with three channels but increase the number of feature maps step-by-step. Figure 3 features the same convolution as Figure 2, but this time, the input volume and our kernel are three-dimensional with depth 3. Note that in this case, since the kernel's depth is equal to the input's depth, the output still has depth 1, as the kernel only has one position in the depth dimension. For the remainder of this paper, all convolutions we discuss have this structure. Of course, this is a restriction, but it makes sense. The three channels we start with are the color channels. Since we want to learn about the structures in the picture, we only want the local connectivity between the width and height dimensions of the picture to matter, not in the color channel dimension. Consequently, we need as many kernels in a layer as we want feature maps in the output. Knowing this, increasing the number of channels is as easy as simply adding more kernels.

One reason to have several output channels is again feature learning. Neural networks are still pretty much a black box for the programmer. One designs it, runs the training process and, evaluates its output on a test set. What exactly the network is computing is hard to understand and subject to current research [52, 36]. Often, when CNNs are explained, each layer and feature map is considered to extract different structures. There are many studies trying to visualize the layers of a neural network to verify that thought, like [51] of Yosinski et al., who also contribute by providing a visualization tool for layers in CNNs. With that explanation, every additional feature map means more space to extract features and can lead to a better understanding of the input. Too many channels, however, can easily lead to overfitting the network to the training dataset.

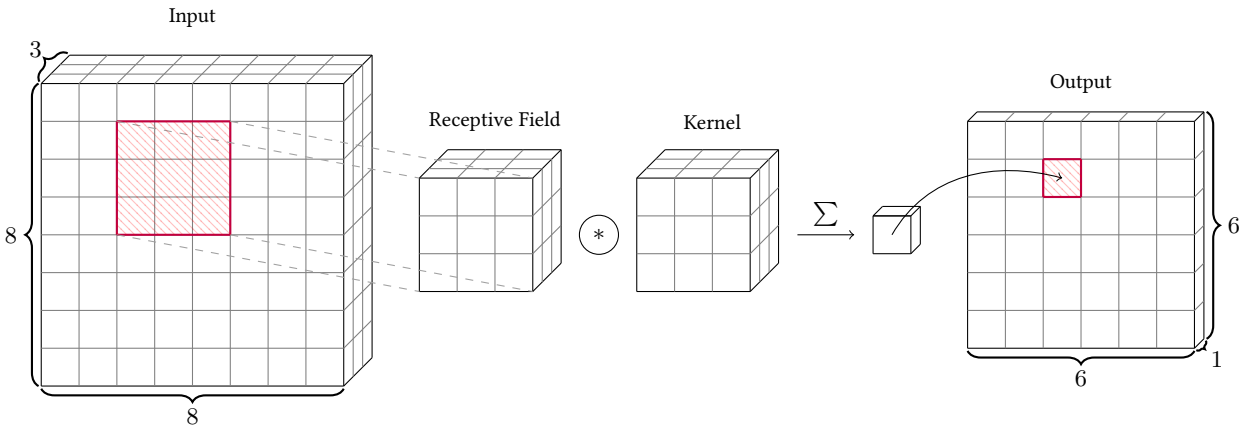


Fig. 3: Depiction of a sample 3D convolution. The output value is $f\left(\sum_{i,j,c=1}^{3} w(i,j,c)\mathcal{R}_{i,j,c} + b\right)$, where $\mathcal{R}_{i,j,c}$ is the value in row i, column j and channel c in the receptive field and $w(i,j,c)$ the weight in the kernel at position (i,j,c). As usual, b is a constant bias and f is an activation function.

3.2.2 Common Layers in CNNs

A CNN does not only make use of convolutional layers. There are other commonly used layers, which we take a look at in this section. The most important one probably being the *pooling layer*. A pooling layer reduces the extent of the height and width dimensions. The motivation for a pooling layer is that the very exact position of a feature is neglectable. Therefore, reducing the amount of information is possible when taking the local neighborhood into account. As a result of the smaller spatial dimensions the network is less expensive to calculate.

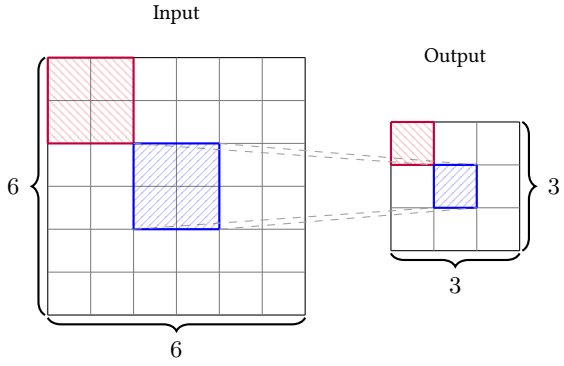


Fig. 4: Depiction of a sample 2D pooling layer with 2×2 kernel and stride 2.

One can describe pooling in terms of a convolution. We can slide a window over the input and look at the local neurons to generate an output value. A popular choice is using a filter of size 2×2 with a stride of 2 in both spatial dimensions. Thus, one uses the value of every neuron exactly once, together with its three nearest-neighbor neurons, to create one output value. This process shrinks the input by a factor of 2 in both width and height, thus reducing the data by 75%.

There are different strategies on how to combine the values in the current receptive field. For example, one could take the mean value of the input neurons as the output value. In practice, however, *mean-pooling* does not yield sufficiently good results. Current research suggests that better results are achieved by *max-pooling*. Instead of taking some combined result of the four pixels, one takes the maximum of the four inputs as the output. Since we described pooling with convolutions, we should mention that after pooling, usually, no activation function is applied.

In Figure 4, we can see an example of a two-dimensional pooling layer with the mentioned parameters. When transferring pooling to three dimensions, a pooling layer over a three-dimensional volume for image processing usually has dimensions $2 \times 2 \times 1$ (This is a contrast to convolutions, where the depth always was equal to the input's depth.). Meaning it works on each feature map separately, like a 2D filter. Larger pooling kernels can also be defined and used, but empirical results show that the information loss for kernels of size 3×3 already hurts performance too much to justify its use. Just like for the convolutional layer, we have edge-cases for pooling layers. For example, it is possible that kernel size, input size, and stride do not play well together, and a receptive field overshoots the end of the input. There is not a general way to solve this problem. One can either define what happens in such a case or one can call these configurations invalid and not allow them to be used, as we did with convolutions.

Using pooling layers often does not hurt the performance of a network and can even improve it. The reason is that the number of neurons is heavily reduced with each layer, allowing faster forwarding. With faster forwarding, we can increase the depth of a network and use more feature maps without blowing up the execution time. Furthermore, after applying a pooling layer, the following kernel effectively has a larger receptive field without additional multiplications and additions. Since pooling layers abstract away a lot of information, they can also prevent overfitting and help to generalize features.

Other layers mainly depend on the use case of the network. If we, for example, want to classify our images, we do not want an image as output but simply probabilities to what class this image belongs. Therefore, one might want to use convolutions to extract features. With these features, one can then apply fully-connected layers to evaluate the features and finally classify the image. This means a CNN can use fully-connected layers as well. However, in case we want an image as output, there are also upsampling layers. An *upsampling layer* is the opposite of a pooling layer. Instead of reducing the dimensions, it increases them by reconstructing the missing information from the available input. There are mainly two different approaches for upsampling layers. Firstly, one can use a transposed convolution, which slides the receptive field of the kernel over the output instead of the input, accumulating the result in the output layer. Secondly, it is possible to increase the dimensions with a known traditional upsampling algorithm, e.g., nearest-neighbor or bilinear, and then apply a convolution afterward.

When using pooling and upsampling layers together, one can combine them to create *cross-connections* between layers. One can pool, for example, with a 2×2 kernel, then do some cheaper convolutions in the reduced spatial dimensionality, and upsample the result again.

After this procedure, the starting and end layers have equal spatial dimensions. Therefore, one can "add" the channels of the starting layer to the feature maps of the end layer. The available extra information can help the reconstruction process. Neural networks that use this kind of cross-connections are counted towards the class of U-Nets, which were introduced by Ronneberger, Fischer, and Brox for biomedical image segmentation [46].

3.3 Implementing Feedforwarding in AnyDSL

The goal of this thesis is to implement a forwarding routine for CNNs in AnyDSL and to apply it. By now, we have sufficient knowledge of CNNs and AnyDSL to tackle the implementation part. The most important component of CNNs is the convolutions. Therefore, we need an efficient way of calculating their result. A direct, straightforward implementation of a convolution like we explained them in section 3.2.1 cannot leverage modern hardware to its advantage. Memory speed often is the bottleneck in today's computers, especially for matrix multiplication algorithms. Thus, optimizing for cache usage is crucial since loading data from cache takes only a few CPU-cycles, while accessing data from memory can easily take more than a hundred cycles. The naïve algorithm mostly ignores the cache architecture, and each time the row in a receptive field is swapped, a whole new cache line needs to be loaded. As this happens several times in every receptive field, for every kernel and convolutional layer, the time spent waiting for data from memory would add up significantly.

By duplicating values of the input and rearranging the kernel weights in memory, we can reduce the problem of convolution to matrix-matrix multiplication. For example, Chellapilla, Puri, and Simard present the im2col rearrangement [8], which creates two matrices to multiply that compute the result of a convolution. With this, we can take advantage of research in the area of fast and efficient matrix multiplications. Programs in all different kinds of fields require matrix-vector and matrix-matrix operations. Basic linear algebra subroutines (BLAS) libraries provide a collection with an easily accessible API to compute such operations efficiently, with memory in mind. On the CPU side, we integrate oneAPI [19] as an example of a BLAS library and use it as a comparison with a native, independent matrix multiplication in AnyDSL in the context of convolutions. On the GPU side, we use cuBLAS [3].

3.3.1 Preliminary Functionality

Starting with the implementation in AnyDSL, we need a way to store images as matrices. The datatype `Matrix` defined by

```
enum MemoryFormat {
  CHW, HWC
}

struct Matrix {
  buf      : Buffer,
  offset   : i64,
  format   : MemoryFormat,
  channels : i32,
  rows     : i32,
  cols     : i32
}
```

is our base construct for the following operations. The buffer `buf` is just a big section of contiguous, allocated memory, with `offset` defining the start of one particular `Matrix` in this buffer. We use this kind of representation to reduce memory allocation overhead. A CNN

usually has many convolutional layers, and their results are not relevant anymore after they have been processed further. Thus, instead of allocating and deallocating memory every time, we can reuse one memory chunk that has enough space to hold everything we need at a given time. In the following code, all functions use this very same concept of buffers with an offset. Regarding the matrix again, we also need to remember the height (`rows`), width (`cols`), and the number of `channels`. Storing images like this is done in two popular memory formats. On the one hand, we have the CHW (channel, height, width) format, in which we store `channels`-many feature maps of size $w \times h$ adjacent to each other. On the other hand, in the HWC format, all feature map values of one spatial position are stored adjacent before the next spatial values begin. The name of these formats also reveals the most efficient iteration order. While an efficient iteration over CHW would be to first iterate over the channels, then the height, then the width, iterating over HWC would put the channels at the end instead of the beginning. To easily access values in a `Matrix`, we can define an accessor, which abstracts the memory format:

```
struct AccM {
  read  : fn(i32, i32, i32) -> f32,
  write : fn(i32, i32, i32, f32) -> ()
}
```

However, this struct does not simplify accessing matrices if we manually need to check for the memory format anyway. Therefore, we have a function that takes a matrix as input and returns the correct accessor filled with a read and a write function:

```
fn @get_mat_acc(m: Matrix) -> AccM {
  fn @get_mat_acc_hwc(m: Matrix) -> AccM {
    AccM {
      read = @|row, col, chn| { bitcast[&[f32]](m.buffer.data)(m.offset +
          ((row * m.cols + col) * m.channels + chn) as i64) },
      write = @|row, col, chn, val| {
          bitcast[&mut[f32]](m.buffer.data)(m.offset + ((row * m.cols + col)
          * m.channels + chn) as i64) = val; }
    }
  }

  fn @get_mat_acc_chw(m: Matrix) -> AccM {
    AccM {
      read = @|row, col, chn| { bitcast[&[f32]](m.buffer.data)(m.offset +
          (row * m.cols + col + chn * m.cols * m.rows) as i64) },
      write = @|row, col, chn, val| {
          bitcast[&mut[f32]](m.buffer.data)(m.offset + (row * m.cols + col +
          chn * m.cols * m.rows) as i64) = val; }
    }
  }

  match m.format {
    MemoryFormat::CHW => get_mat_acc_chw(m),
    MemoryFormat::HWC => get_mat_acc_hwc(m)
  }
}
```

As cross-connections are a popular concept in CNNs, we need an efficient way of supporting them. For this purpose, the function above does not suffice. Thus, we write a second function to get a concatenated accessor. Instead of reordering the memory such that all matrices have the same memory format and are positioned adjacent in the memory, we can combine two accessors and use them as if it was one matrix:

```
fn @get_cat_mat_acc(acc1: AccM, acc2: AccM, channels1: i32) -> AccM {
  AccM {
    read = @|row, col, chn| { if chn < channels1 { acc1.read(row, col, chn) }
        else { acc2.read(row, col, chn - channels1) } },
    write = @|row, col, chn, val| { if chn < channels1 { acc1.write(row, col,
        chn, val); } else { acc2.write(row, col, chn - channels1, val); } }
  }
}
```

Using this method, we can concatenate several matrices without much additional computational effort. AnyDSL allows the definition of custom iteration functions. Since we need to iterate over matrices more than once when implementing the forward propagation, it seems reasonable to implement a matrix iterator. Like with the accessors, we can abstract the way the matrix is stored in memory by writing a function that always uses the more efficient iteration order:

```
fn @iterate_matrix(body: fn(AccM, f32, i32, i32, i32) -> ()) {
  let loop_chw = iterate_matrix_chw(body);
  let loop_hwc = iterate_matrix_hwc(body);
  fn @loop(mat: Matrix) -> () {
    match mat.format {
      MemoryFormat::CHW => loop_chw(mat),
      MemoryFormat::HWC => loop_hwc(mat)
    }
  }
  loop
}
```

The values this function provides us in the body are a matrix accessor, the value of the matrix at the current position, and the current row, column, and channel. If we need a specific iteration order, the `iterate_matrix_chw` as well as the `iterate_matrix_hwc` functions are both globally accessible. They behave as one would expect, with `CHW` having channels as the outermost loop, width as the innermost loop, and `HWC` having height as the outermost loop, channels as the innermost loop. Every iteration function mentioned here has a parallel counterpart by adding `_par` to its name that exchanges the outermost `range()` call with a `parallel()` call.

3.3.2 The Matrix Multiplication

Having defined the basic matrix accessors, we can think of implementing a matrix multiplication. Again, the naïve algorithm is not very useful and efficient, but we list it here, as it can help understanding the framework:

```
fn @matmul(a: Matrix, b: Matrix, buf: Buffer, off: i64) -> Matrix {
  let m = make_matrix(buf, off, MemoryFormat::HWC, 1, a.rows, b.cols);
  let a_acc = get_mat_acc(a);
  let b_acc = get_mat_acc(b);

  for m_acc, _v, r, c, chn in iterate_matrix_par(m) {
    m_acc.write(r, c, chn, 0);
    for i in range(0, a.cols) {
      let updated_v = m_acc.read(r, c, chn) + a_acc.read(r, i, chn) *
          b_acc.read(i, c, chn);
      m_acc.write(r, c, chn, updated_v);
    }
  }
  m
}
```

The function takes as arguments two matrices `a` and `b` and a memory location for the result, defined by a buffer and an offset. First, the output matrix is created, along with two accessors for the input matrices. In this simple algorithm, we iterate over the output matrix and initialize every value with 0. We then use the definition of matrix multiplication to compute the current position in a third loop. Note that since we do not have any memory conflicts, we can use the parallel matrix iteration, and we also do not care about the iteration order and can always use the more efficient one. However, even now, we do not gain much, if anything at all, compared to the straightforward convolution. With closer inspection, one can see, for example, that each row of the input matrix `a` is loaded into cache several times if the cache is not big enough to hold the whole row at once (which often is the case). Also, we are accessing the values of `b` by column, always loading a whole row into cache and only using one value. Adding to this, we need to reorganize our input before even using matrix multiplication for convolution. It is self-evident that we need a better matrix multiplication algorithm.

Theoretically, the runtime of the approach above is $\mathcal{O}(n^3)$ with $n \times n$ being the size of the matrices. It is possible to achieve theoretical runtimes of less than $\mathcal{O}(n^{2.373})$ [2]. However, algorithms with an exponent that low often have large constant factors in return. Therefore, most of them are not used in practice. Strassen presents an algorithm in [48] that partitions the input and output matrices into four smaller, equally sized block matrices to calculate the product with one less multiplication than the naïve algorithm would. Doing this recursively until the matrices are only numbers results in a better asymptotic complexity than $\mathcal{O}(n^3)$. This algorithm is one of the few that is used in practice. An overview brought together by Bläser provides proofs and concepts of other theoretical, more efficient matrix multiplication algorithms in [4].

Nonetheless, since the algorithms that are used in practice are mainly beneficial when working with square matrices, and we have strongly rectangular matrices, we stick with the cubic algorithm. To make using it feasible at all, we need ideas to optimize its cache usage. As described before, the problem in the naïve approach is that cache lines are overwritten, although the values are needed at a later point again. Addressing this is possible by *blocking* (also called *tiling*) the matrices into smaller ones and computing their results to sum them up to achieve the complete solution. With a tiled algorithm, one can choose the block size with respect to the cache size, such that a cache line is only overwritten when the values are not needed anymore. Blocked algorithms and their efficiency are thoroughly discussed in [27] by Lam, Rothberg, and Wolf. The authors also present a blocked matrix multiplication algorithm, which we implement in AnyDSL to measure its performance. The implementation in Artic looks like this:

```
fn @matmul_cpu(a: Matrix, b: Matrix, buf: Buffer, off: i64) -> Matrix {
  let mat = make_matrix(buf, off, MemoryFormat::HWC, 1, a.rows, b.cols);

  let a_acc = get_mat_acc(a);
  let b_acc = get_mat_acc(b);
  let m_acc = get_mat_acc(mat);

  let c_size = 64; // Blocking over columns of b
  let i_size = 16; // Blocking over columns of a
  let mut m : [f32 * 64]; // Needs "c_size"-many elements
```

```
  for cc in range_step(0, mat.cols, c_size) {
    for ii in range_step(0, a.cols, i_size) {
      for r in range_step(0, mat.rows, 1) {
        for j in range(cc, min(mat.cols, cc + c_size)) {
          if ii == 0 { // Initialize all values first written to with 0
            m(j) = 0;
          } else {
            m(j) = m_acc.read(r, c + j, 0);
          }
          for i in range(ii, min(a.cols, ii + i_size)) {
            m(j) += a_acc.read(r, i, 0) * b_acc.read(i, c + j, 0);
          }
          m_acc.write(r, c + j, 0, m(j));
        }
      }
    }
  }
  mat
}
```

First, we create all necessary accessors and set the block sizes. Then, the two blocking loops follow, which iterate over the columns of the two input matrices in step sizes equal to the block sizes. In the block defined by these two outermost loops, smaller matrix multiplication is done, and, thus, we have three more loops iterating over every value in this block. The one if-statement initializes the values we are currently computing either by 0 if it is the first time we encounter this output location or by reading the current value if we encountered it before. To not overshoot in the inner loops, they are only iterating to the minimum over the end of the block and the end of the matrix. Using the parameters `c_size` and `i_size` we can adapt the algorithm to the cache size available on the system.

Note that AnyDSL only supports magic numbers in the context of arrays, even when their size would be known at compile-time. Therefore, we set `m`'s length manually to `c_size`.

With this code, we turned many cache misses into cache hits. Nonetheless, there are optimizations to be done still. First and foremost, modern hardware can use *single instruction multiple data (SIMD)* instructions to perform the same operation on different data in parallel, provided the software is written accordingly. Artic allows us to easily *vectorize* the matrix multiplication algorithm above by exchanging a range loop with a vectorization call (Vectorization means allowing the use of SIMD instructions.). Since every value calculated needs its own container in the vectorized loop, the structure of the implementation is already helpful, and we can swap the fourth loop (the `j` loop) for a vectorization call like this:

```
vectorize(64, @|j| {
  if c + j < mat.cols {
    // ...
  }
});
```

The vectorization also added another if-statement since we do not want to overshoot the matrix and cannot stop the loop earlier as we did before. However, the efficiency of the automatic vectorization by the Artic compiler is hindered by branches in the `vectorize` body. For example, if, within the same batch, different branches are taken, this can lead to worse performance [35]. Thus, we need reduce the number of branches as much as possible. Luckily, this is not too hard because we can split all of them manually and only vectorize the parts where it makes sense. Applying this, we split the `ii` loop into three parts:

1. `ii == 0`: To initialize `m(j)` with 0; `j` loop can be vectorized,
2. The part where a whole block of `i_size` fits into the matrix, to compute with vectorization,
3. The remaining part which is less than `i_size` to calculate unvectorized.

The if-statement added by introducing vectorization (`if c + j < mat.cols`) only needs to be split into two parts. They are more or less the same as 2. and 3. from the `ii` loop:

1. The part where a whole block of `c_size` fits into the matrix, to compute with vectorization,
2. The remaining part which is less than `c_size` to calculate unvectorized.

Splitting this up by hand makes the whole function not easily readable, but we removed all the branches in the vectorized loop.

Instead of initializing every output value by 0, we can take advantage of the fact that we only need the matrix multiplication for the convolution. Therefore, we can pass a higher-order function

```
biases: fn(i32) -> f32
```

which, given the number of the kernel we are currently processing returns the corresponding bias. We can then add the bias before the calculation with

```
m(j) = biases(r); // instead of: m(j) = 0;
```

since every row `r` represents one kernel, as explained later with Figure 6. Doing this, we can spare the iteration after the matrix multiplication to add a bias to each value in the output. If a convolution does not use a bias, we can pass a function returning 0 for every input. The partial evaluator replaces this by 0 automatically at compile-time.

We can do the same for the application of the activation function. As discussed in section 3.2, after each convolution, an activation function is applied. With the same reasoning as for the bias, we can apply this function the moment we know the value at a specific position in the output would be the final value. We can pass a function

```
act_fn: fn(f32) -> f32
```

to the matrix multiplication and apply it in the innermost loop, the `ii` loop in the last iteration of the outer loops. Like before, if the values should stay the same, one can pass an identity function `@|v| -> v`, which is replaced at compile-time by simply the value itself without a function call. Once again, this means we do not need to iterate over the output to apply the activation function. Specializing the matrix multiplication like this is one advantage we have when implementing it ourselves instead of using a BLAS library which is far less flexible. The function's signature looks like this after adding both high-order functions to it:

```
fn @matmul_cpu(a: Matrix, b: Matrix, biases: fn(i32) -> f32, act_fn: fn(f32)
    -> f32, buf: Buffer, off: i64) -> Matrix
```

3.3.3 The im2col Rearrangement

We are now able to efficiently multiply matrices on the CPU using the native, cache-efficient algorithm written in Artic. Next, we need to create the matrices we want to multiply. For the im2col-based convolution [8], we transform the image matrix into a matrix in which each

column contains one receptive field. Consequently, its number of columns is equal to the number of receptive fields that fit into the input, like shown in Figure 5.

Performing one convolution is then achieved by multiplying this matrix from the right with a *flattened kernel*, seen in Figure 6. An advantage of the flattened kernels is that we can already load them into memory like this and do not need to rearrange anything. Moreover, we can turn several of them into a flattened kernel matrix, where each row contains one flattened kernel. Having this layout allows us to use one matrix multiplication for all convolutions in one convolutional layer. Since all kernels in one layer have the same window size, we only need to go through the im2col process once for all convolutions in a layer. Thus, the time consumption of copying in this process is neglectable but, nonetheless, should still be optimized.

We can transfer the rearrangement to three-dimensional volumes from the 2D examples in the figures. It works precisely the same way, only that we need to consider the memory format again. Both the flattened kernels and im2col matrix need to be in the same memory format such that the weights are correctly multiplied with the corresponding feature map value.

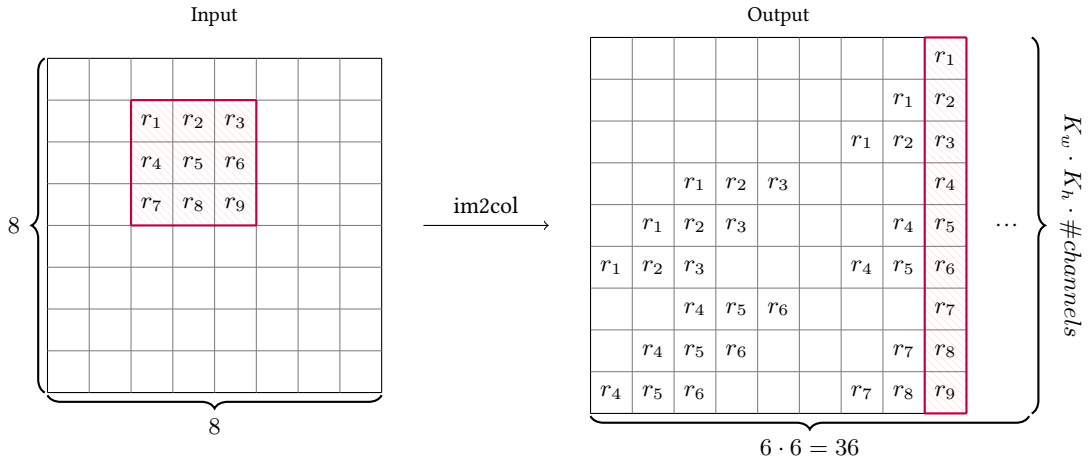


Fig. 5: 2D Sketch of rearranging the values of the image matrix to the im2col matrix with a 3×3 window. Only values in the marked receptive field are named in the output. K_w and K_h are the kernels' width and height, respectively.

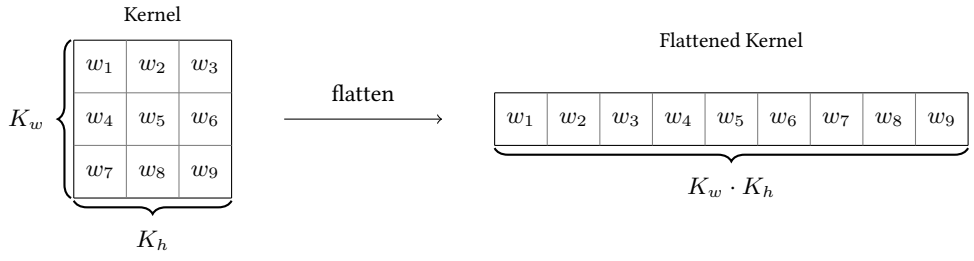


Fig. 6: 2D Sketch of flattening a 3×3 kernel.

With these considerations done, we can implement the im2col rearrangement like this:

```
fn @im2col_cpu(ksize: i32, out_width: i32, out_height: i32, in_width: i32,
    in_height: i32, channels: i32,
        img_acc: AccM, padding: i32, buf: Buffer, off: i64) -> Matrix {
   let img_s = (out_width as i64) * (out_height as i64);
   let ksize_sq = (ksize as i64) * (ksize as i64);

   let bc = bitcast[&mut[f32]](buf.data);
   let res_mat = make_matrix(buf, off, MemoryFormat::CHW, 1, channels *
      ksize_sq as i32, img_s as i32);

   for chn in parallel(0, 0, channels) {
      for r in range(0, out_height) {
         for c in range(0, out_width) {
            let mut base_row = (chn as i64) * (img_s * ksize_sq);
            let     base_col = r * out_width + c;
            let p_row = r - padding;
            let p_col = c - padding;

            for y in unroll(0, ksize) {
               for x in unroll(0, ksize) {
                  let h = p_row + y;
                  let w = p_col + x;

                  if unlikely(h < 0 || w < 0 || h >= in_height || w >=
                      in_width) {
                     bc(off + base_row + (base_col as i64)) = 0; //
                        zero-padding
                  } else {
                     bc(off + base_row + (base_col as i64)) = img_acc.read(h,
                        w, chn);
                  }

                  base_row += img_s;
               }
            }
         }
      }
   }
   res_mat
}
```

Since this is a big chunk of code, let us look at it piece by piece. First, the image size, meaning the number of values in one output feature map, is stored in a local variable `img_s`. Later in this function, it is needed as an offset for the channel values in the im2col matrix. Next, we use a `bitcast` to interpret the given buffer as a float buffer, and create the result matrix `res_mat`. Iterating over the input channels and the output's size, we can fill the im2col matrix step-by-step. In the end, one column contains the window values of all feature maps. The first K^2 values are from one feature map, the following K^2 from the next one, and so on. With this layout, we have something similar to a CHW format.

At first sight, the computation might seem complicated, but the underlying concept is simple. If we want to fill position (r, c) in the matrix our convolution should produce, then the corresponding column in the im2col matrix is given by $r \cdot out_width + c$ since we have as many columns in the im2col matrix as we have pixels in the output. Looking at the code, one can see that this is precisely what the variable `base_col` stores. `base_row` contains the corresponding row at which the values for the current feature map start.

We now know the column we want to fill for this position, but we still need to know the values to fill it. By reconstructing the receptive field of the coordinates (r, c) in the input, we can compute them. To this end, we first need to account for the padding p by subtracting it from r and c as it moves the origin from its original location. The value found at that position is where the receptive field starts. Thus two loops follow, iterating over the kernel size to fill the column `base_col` of the current feature map. By adding the offset in the receptive field to the padded row and column position, one can calculate the coordinates in the input image. If, however, this position is invalid, it has to be a padded value. We always set it to 0 in this case, which means we currently use zero-padding. It is the most common type of padding, but since one can easily exchange this constant for another constant or function call, it should not pose a problem to add support for other padding types. After each loop iteration, the next value should be written one row below. To achieve this, we add `img_s` to `base_row` to stay in the same column, just one row below. A restriction we make here to keep the routine simpler is that the stride has to be one. In each iteration, we move the kernel by only one step in the x or y direction. By changing this and adjusting output dimensions, it would be possible to allow other strides as well.

Taking a look at the efficiency of this function, we can observe that the outermost iteration is a parallel call over the input channels. Thus, an input in CHW format is more cache efficient. As the matrix produced by performing convolutions with this algorithm also yields a matrix in CHW format, this is not a problem. Using AnyDSL's partial evaluation, the two innermost loops that are iterating over the receptive field's size can be unrolled since the kernel size usually is small and known at compile time. By calling `unlikely` around a condition, we tell the compiler that it is not likely for this condition to occur, thus helping the branch predictor. Optimizing the im2col function is not the bottleneck of the propagation through the network since it only needs to be done once for many convolutions in a single layer. The results presented in chapter 5 support this claim by showing that most of the time is spent multiplying matrices, although we already optimized it.

3.3.4 The Convolution Function

Having implemented everything we need to compute a convolution, we can write a convolution function:

```
fn @conv2d(ksize: i32, out_channels: i32, flattened_kernels: Matrix, biases:
    fn(i32) -> f32, act_fn: fn(f32) -> f32, img_mat: Matrix, buf: Buffer, off:
    i64, off_res: i64) -> Matrix {
  cat_conv2d(ksize, out_channels, img_mat.channels, flattened_kernels,
      biases, act_fn, img_mat.rows, img_mat.cols, get_mat_acc(img_mat), buf,
      off, off_res)
}

fn @cat_conv2d(ksize: i32, out_channels: i32, in_channels: i32,
    flattened_kernels: Matrix, biases: fn(i32) -> f32, act_fn: fn(f32) -> f32,
    in_rows: i32, in_cols: i32, img_acc: AccM, buf: Buffer, off: i64, off_res:
    i64) -> Matrix {
  let padding    = ksize / 2; // insize == outsize
  let out_width  = (in_cols + 2 * padding - ksize) + 1;
  let out_height = (in_rows + 2 * padding - ksize) + 1;

  let im2col_mat = im2col(ksize, out_width, out_height, in_cols, in_rows,
      in_channels, img_acc, padding, buf, off);
  let prod = matmul(flattened_kernels, im2col_mat, biases, act_fn, buf,
      off_res);
  make_matrix(prod.buf, prod.offset, MemoryFormat::CHW, out_channels,
      out_height, out_width) // Reshape matrix for future use
}
```

`conv2d` is a simple wrapper to perform a convolution with a matrix without turning it into a matrix accessor manually. It only calls `cat_conv2d`, which computes a convolution in mainly two steps. First, it calls `im2col`. Then it multiplies the result with the flattened kernels, which it expects to have the correct format already.

The padding P is restricted to $P = \lfloor \frac{K}{2} \rfloor (= \frac{K-1}{2}$, if K is odd), which means every convolution has the same spatial input dimensions as output dimensions, as we discussed in section 3.2. However, changing the function to take it as a parameter instead, making it more flexible, is not a problem. We decided to make this restriction because all convolutions we need have a padding of $\frac{K-1}{2}$. Instead of passing it every time, we could just perform that calculation once here, not having to do it in every convolution call. As the name `cat_conv2d` suggests, one can call it with a concatenated matrix accessor and, thus, can use it for cross-connections between layers.

3.3.5 Pooling and Upsampling Layers

As mentioned in section 3.2.2, to propagate a convolutional neural network, we also need to implement other essential layers. For our implementation of the pooling layer, the configurations where the last receptive field only fits partly on the input are invalid and not allowed. We assign the task to check the configuration to the caller of the pooling function. By omitting this check, the pooling function can be more efficient because if the caller knows that it will be valid, they do not need to check for it. We can then implement it by looping over the output matrix in parallel:

```
fn @pool_cpu(img_mat: Matrix, size: i32, pooling_fn: fn(fn(i32) -> f32) ->
    f32, buf: Buffer, off: i64) -> Matrix {
  let res_mat = make_matrix(buf, off, img_mat.format, img_mat.channels,
      img_mat.rows / size, img_mat.cols / size);
  let img_acc = get_mat_acc(img_mat);

  for r_acc, _v, row, col, chn in iterate_matrix_par(res_mat) {
    let mut v : [f32 * 4]; // Needs at least size_squared-many elements.
    fn @values(idx: i32) -> f32 {
      v(idx)
    }

    for r in unroll(0, size) {
      for c in unroll(0, size) {
        v(r * size + c) = img_acc.read(size * row + r, size * col + c, chn);
      }
    }

    let val = pooling_fn(values);
    r_acc.write(row, col, chn, val);
  }

  res_mat
}
```

Note that our implementation of the pooling layer does not allow overlapping receptive fields, i.e., the stride is always equal to the window size. By having a function given to the layer, we can compute arbitrary pooling operations with the same method. As the size is usually known at compile-time, the iterations over it can be unrolled. This function does not support in-place pooling since it uses a parallel iteration over the matrix, which might lead to data races. For in-place pooling, we provide a second, similar pooling function that works without the parallel call.

Again, one has to mention that AnyDSL does only support magic numbers in the context of arrays, even when the size is known at compile-time. Therefore, v has a fixed length of 4 here since 2×2 kernels are the most common ones. If larger kernels are needed, this number needs to be changed accordingly.

The opposite of the pooling layer, an upsampling layer, can be implemented in the same way by looping over the output matrix in parallel and filling it with values. As an example of an upsampling layer, we implement nearest-neighbor upsampling because it is fast, often used, and other interpolation-based approaches do not seem to offer better accuracy, as, for example, a comparative study by Kolarik, Burget, and Riha has shown [25].

```
fn @nearest_cpu(img_mat: Matrix, buf: Buffer, off: i64) -> Matrix {
   let res_mat = make_matrix(buf, off, img_mat.format, img_mat.channels,
       img_mat.rows * 2, img_mat.cols * 2);
   let img_acc = get_mat_acc(img_mat);

   for res_acc, _v, row, col, chn in iterate_matrix_par(res_mat) {
      res_acc.write(row, col, chn, img_acc.read(row / 2, col / 2, chn));
   }
   res_mat
}
```

Since we want a high-performance implementation, there also is another way that, in theory, sounds more efficient. Instead of upscaling at once by iterating over the whole matrix, we can create an upsampled accessor. With this approach, we do not need to copy all the values to another memory location but can calculate the upsampled value as we need it.

```
fn @nearest_acc(img_mat: Matrix) -> AccM {
  let acc = get_mat_acc(img_mat);

  AccM {
    read  = @|row, col, chn| { acc.read(row / 2, col / 2, chn) },
    write = @|row, col, chn, val| { acc.write(row / 2, col / 2, chn, val) }
  }
}
```

For the pooling layer, however, it does not seem useful to create a pooled accessor in a similar way. The reason is, when iterating a matrix, we optimized for cache usage. Using a pooled accessor, we need to look at the whole receptive field to calculate the value, which might destroy all the assumptions about the cache and yield worse access patterns. Thus, it seems more promising to do pooling explicitly. With this work done, implementing a neural network model with a forwarding routine should not pose a problem on the CPU side, at least.

3.3.6 Other Backends

For the GPU implementation, we need to provide similar functions using similar ideas. When having multiple backends, we also want to swap between them without much effort. Thus, we first explain our approach of choosing a backend and then present the code snippets for the GPU side.

As the core functionality on CPU and GPU is the same, only functions computing values need to be changed. For example, the matrix representation, accessors, iterators can all be used on the GPU as well. We can also keep the two convolution functions since they only call the im2col and matrix multiplication routines, as long as we ensure that the correct functions are called. Thus, we provide some internal mappings for the neural network with the following struct:

```
struct NNInt {
   matmul : fn(Matrix, Matrix, fn(i32) -> f32, fn(f32) -> f32, Buffer, i64)
       -> Matrix,
   im2col : fn(i32, i32, i32, i32, i32, i32, AccM, i32, Buffer, i64) ->
       Matrix,
   pool : fn(Matrix, i32, fn(fn(i32) -> f32) -> f32, Buffer, i64) -> Matrix,
   add_element_wise : fn(Matrix, AccM, Buffer, i64) -> Matrix,
   chw_to_hwc : fn(Matrix, Buffer, i64) -> Matrix
}
```

Passing an instance with the correct mappings through the forwarding allows every function to be called on the current backend. For the CPU, a mapping could look like this:

```
fn @get_cpu_nn() -> NNInt {
   NNInt {
      matmul  = matmul_cpu,
      im2col  = im2col_cpu,
      pool    = pool_cpu,
      add_element_wise = add_element_wise_cpu,
      chw_to_hwc = chw_to_hwc_cpu
   }
}
```

Seeing this, it is also clear what functionality we need to implement for the GPU side to be working. We ignore the `add_element_wise` and `chw_to_hwc` implementations as they are self-explanatory and trivial to implement.

We spent a considerable amount of time optimizing the matrix multiplication on the CPU side. Therefore, we decided for the GPU implementation to be rather basic and used the naïve $\mathcal{O}(n^3)$ algorithm as a proof-of-concept and comparison. In AnyDSL one can create an accelerator for a specific backend that provides said backend's functionality. For our case, we want to use CUDA, and with `cuda_accelerator(device)` we can create an accelerator using CUDA as the backend for the given device. Launching kernels is possible by defining the block and grid sizes and then calling execute on the accelerator. With `accelerator.sync()`, we can join all started threads and wait for them to finish. With this knowledge, we can implement the matrix multiplication like this:

```
fn @matmul_gpu(a: Matrix, b: Matrix, biases: fn(i32) -> f32, act_fn: fn(f32)
    -> f32, buf: Buffer, off: i64) {
   let acc = cuda_accelerator(0);

   let c = make_matrix(buf, off, MemoryFormat::HWC, 1, a.rows, b.cols);

   let a_acc = get_mat_acc(a);
   let b_acc = get_mat_acc(b);
   let c_acc = get_mat_acc(c);

   let threads_x = 32;
   let threads_y = 32;
   let block = (threads_x, threads_y, 1); // 32 * 32 * 1 = 1024 Threads/Block
   let grid = (round_up(c.cols, threads_x), round_up(c.rows, threads_y), 1);
```

```
    for work_item in acc.exec(grid, block) {
        let row = work_item.bidy() * work_item.bdimy() + work_item.tidy();
        let col = work_item.bidx() * work_item.bdimx() + work_item.tidx();

        if row < c.rows && col < c.cols {
            let mut v = biases(row);
            for i in range(0, a.cols) {
                v = v + a_acc.read(row, i, 0) * b_acc.read(i, col, 0);
            }
            c_acc.write(row, col, 0, act_fn(v));
        }
    }
    acc.sync();
    c
}
```

First, we create an accelerator and the result matrix `c`. We take accessors to all our matrices, define the grid and block size, and iterate over the matrices. Since we are blocking our matrix on the GPU inherently, we need to round up the grid size to a multiple of the block size. Consequently, it can happen that the calculated row and column overshoot the input dimensions. Thus, the if-statement is necessary to check whether the current row and column are valid. If one of them is not, we know that the block had to be rounded up, and we can ignore this thread. However, this does not matter much as GPUs are optimized for this kind of if-statements directly after the kernels are launched. Adding biases and applying the activation function can be done exactly as on the CPU side. At the end of the function, we wait for the launched kernels to finish before finally returning the product matrix.

In the same way, we can transfer the pooling function to the GPU. At the beginning we need to create an accelerator and define the block and grid sizes which can be done like this:

```
let acc = cuda_accelerator(0);
let threads_x = 32;
let threads_y = 32;
let block = (threads_x, threads_y, 1);
let grid = (round_up(res_mat.cols, threads_x), round_up(res_mat.rows,
    threads_y), res_mat.channels);
```

When executing the kernels, we need to compute the column, row, and channel from the block indices and check whether this is a valid position in the volume. If it is, we can perform the same operations as we did on the CPU. Finally, we synchronize our kernels again before returning the pooling result.

Last but not least, we need to have a GPU implementation of the im2col routine. For this, the way we computed it on the CPU is beneficial already, as we can transfer it 1-to-1 to the GPU. The minor changes are the essential ones. We need the CUDA accelerator and block and grid sizes to iterate over like in the two cases before:

```
let acc = cuda_accelerator(0);
let i_size = 32; // Image (w*h) dimension
let c_size = 8; // Channel dimension
let block = (i_size, c_size, 1);
let grid = (round_up(img_s, i_size), round_up(channels, c_size), 1);
```

As we do not iterate over the input matrix explicitly anymore, but over the blocks on the GPU, we need to calculate the `r` and `c` that we used on the CPU. However, since we consider one column of the im2col matrix to be one receptive field on the input, this calculation is

straightforward. We need the column in the im2col matrix we are currently processing, which is given by

```
let idx = work_item.bdimx() * work_item.bidx() + work_item.tidx();
```

and then the corresponding input image's `r` and `c` are

```
let r = idx / out_width;
let c = idx % out_width;
```

After launching the kernels, we need to wait for them to finish. The rest of `im2col_gpu` is the same as on the CPU. These implementations suffice to execute a neural network model on the GPU.

Since we now have a way to swap mappings of specific functions, the usage of BLAS libraries for the matrix multiplication is as easy as changing the matrix multiplication of the mappings instance but leaving the rest the same. Thus, we implement two more matrix multiplications, one using oneAPI, and one using cuBLAS. Since these are matrix multiplications, we need to iterate over the results again, to add the bias and apply the activation function afterward. These implementations are the ones that we use as a comparison baseline in chapter 5.

The oneAPI function wrapper, for example, looks like this:

```
fn @matmul_oneapi(a: Matrix, b: Matrix, biases: fn(i32) -> f32, act_fn:
    fn(f32) -> f32, buf: Buffer, off: i64) {
   let c = make_matrix(buf, off, MemoryFormat::HWC, 1, a.rows, b.cols);

   mkl_blas_mm_mult(/*m*/ a.rows, /*n*/ b.cols, /*k*/ a.cols,
      /*a*/ bitcast[&   [f32]](&(a.buf.data(4 * a.offset))), /*lda*/ a.cols,
      /*b*/ bitcast[&   [f32]](&(b.buf.data(4 * b.offset))), /*ldb*/ b.cols,
      /*c*/ bitcast[&mut[f32]](&(c.buf.data(4 * c.offset))), /*ldc*/ c.cols);

   for acc, v, row, col, chn in iterate_matrix_par(c) {
      acc.write(row, col, chn, act_fn(v + biases(row))); // 1 channel per
          row, thus access row-th bias
   }
   c
}
```

4. Ray Tracing and Denoising

As an application and demonstration of usability of the implemented forward propagation, this chapter covers the integration of our library into Rodent, a ray-tracing library written in AnyDSL. To this end, we start by discussing the process of rendering scenes via ray tracing and the problems that arise. Having done that, we use a neural network to achieve better rendering results.

4.1 Ray Tracing

The term "ray tracing" refers to a technique used in rendering to model the way light is transported. For a long time, rasterization has been the standard rendering technique for real-time applications. In rasterization, light is baked into the objects' textures, which means its effects are pre-calculated. Displaying the effect of pre-calculated light is obviously computationally less expensive but comes at a price. For example, shadows are inaccurate, and it is hard to make them look clean. Additionally, rasterization has problems when dealing with (semi-)transparent materials and their refractions as there is no known algorithm to smoothly integrate them into the rasterization pipeline. Reflections suffer from the same problem.

These problems can be solved by modeling light as rays, which produces sharp shadows. With the refraction and reflection indices, the computation of (semi-)transparent and reflective materials does not pose a problem either, mathematically at least. However, ray tracing comes with its own set of problems. First and foremost, it is expensive to compute the effects of light by tracing rays, which was the reason for it not being used in real-time applications for a long time. Only recently has it become standard to have hardware-acceleration support for ray tracing on modern GPUs. The second problem remains, which are numerical stability and convergence rates of ray tracing.

In a published paper, Kajiya states that every rendering technique essentially tries to solve the rendering equation. The author presents it in [24], and it is given by

$$L_o(x, \omega_o) = L_e(x, \omega_0) + \int_{\Omega_+} f_r(\omega_i, x, \omega_o) L_i(x, \omega_i) cos\theta_i \, d\omega_i,$$

where the total outgoing radiance L_o at position x in direction ω_o is expressed as the sum of the emitted radiance L_e at position x in direction ω_o and the reflected radiance. The reflected radiance is the integral of the incident radiance L_i at that position from all directions $\omega_i \in \Omega_+$ multiplied with a reflection function f_r defining what fraction of the incoming light is reflected in direction ω_o. Additionally, the contribution of the incoming radiance is weighted by the incident angle θ_i.

By shooting rays at the scene through pixels and tracing them, a contribution color is accumulated for each pixel and ray. One such ray is a sample, and ray tracing wants to approximate the integral by taking such samples. For suitable samples, the law of large numbers ensures that the approximation improves as the number of samples increases. However, every shot ray is costly to compute as well. Finding a good balance between cost and image quality is a hard problem. Especially in real-time applications, the cost needs to be as low as possible. The noise emerging from lower sample counts, and general numerical errors, can be an annoyance and impact image quality badly. Approaches to remove that noise have recently consisted of

using neural networks and showed promising results [20, 7]. Thus, for the rest of this chapter, we focus on integrating denoising neural networks into an existing renderer using the code we presented in the preceding chapters.

4.2 Integration Into Rodent

Open Image Denoise [20] is a pre-trained denoising library that can be integrated easily in existing renderers through a provided C/C++ API. The library accepts as input three RGB images, namely a rendered image, a *normal image*, and an *albedo image*. Since Rodent was not built with the integration of a denoiser in mind, it only creates the rendered noisy image. Thus, it is necessary for us to generate two more images during the rendering process if we want to integrate OIDN. Rodent allocates a buffer of size `3 * width * height * sizeof(float)` for the render in C++. We can create two more buffers of the same size, which we pass to Artic in the same manner.

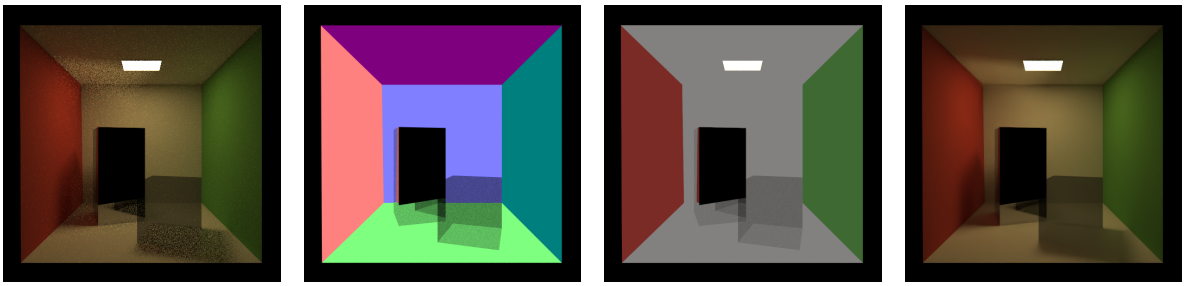
In Rodent, the value for a pixel is accumulated over time by shooting primary and secondary rays that store their past hits by saving the current contribution color. When the rays terminate, they are added to the image buffer, and new rays can be shot. Primary rays are the rays coming directly from the camera and the ones being reflected on the surface through global illumination. Secondary rays are shadow rays, contributing through direct illumination only. Since the normal and albedo images are light-independent, we do not care about secondary rays. However, since there can be (semi-)transparent materials, like glass, in a scene, which need to be sampled, it does not suffice to render a normal and albedo image with one sample only. Instead, we also need to accumulate normal and albedo colors over several samples to have them converge to their real values. This means we need to remember the contributions to these auxiliary images in the rays' memory as well. Whenever a primary ray hits a surface, the renderer's `on_hit` function is called, returning the hits' color. By default, this call returned one color, which we need to change to return three colors to account for the two auxiliary images. To compute the color of the auxiliary images, we need to check if the hit material is specular. If so, then this hit should not be used for coloring. By definition, we want the first non-specular hit to define the normal and albedo color. Otherwise, we can calculate the normal color with the formula

$$v = 0.5n + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix},$$

where v is the resulting RGB color, n is the normalized normal of the hit's surface, and "$+$" is the element-wise vector addition. Note that this formula can never produce the color black, i.e., 0. Because we want to contribute to auxiliary images only once per shot ray, we can initialize the normal color as black and only contribute if it is still black. Contributing to the albedo image works under the same conditions. For every bidirectional scattering distribution function (BSDF), we added a function `albedo`. Calling this functions returns the corresponding albedo color, which depends on the BSDF the material has, e.g., a purely diffuse BSDF's albedo is defined by $a = \frac{1}{\pi}k_d$ and a Phong BSDF's albedo by $a = \frac{1}{\pi}k_s$. Then, if the material itself is emissive, this color is multiplied by the light source's intensity.

By adding these computations and changing some function signatures, the albedo and normal buffers are filled with the correct values during the rendering. Denoising with OIDN is now possible by passing the corresponding pointers and an output pointer to the C++ API. It is important to note that the color values first need to be divided by the sample count. The noisy and albedo images additionally need to be gamma-corrected, creating some overhead

when denoising live during the rendering. Figure 7 showcases the rendered images with this implementation in a sample scene.



(a) noisy **(b)** normal **(c)** albedo **(d)** denoised

Fig. 7: Sample images rendered in Rodent. The scene is a cornell box, with the tall box being a mirror and the cube being transparent.

With OIDN live denoising working in Rodent, the next step is integrating the forward propagation presented in chapter 3. First, we need a trained model and implement a forwarding routine. The model we use is one of the models presented in [23]. In particular, its design is very close to OIDN's. It consists of 16 convolutions (all kernel size 3×3), 4 pooling layers (kernel size 2×2) and 4 upsampling layers. The results of three of the pooling layers are stored in memory to perform cross-connections with the upsampling path of the network. Like in OIDN, all the upsampling layers use nearest-neighbor upsampling followed by a convolution. All convolutions have a bias and use a leaky ReLU as their activation function, except for the last one, which has no activation function.

We can write a higher-order function, setting all necessary parameters for the forwarding, given the image's dimensions, kernel weights, and biases, and returns a forward function. It takes as input five buffers: The noisy, albedo, and normal images, a buffer to store temporary data in, and an output buffer. The necessary size for temporary data is returned along with the forwarding function such that the caller can make sure it does always suffice. As discussed in section 3.3, we chose this design so that we only need to allocate memory once and can reuse it during the rendering. To access the biases and the flattened kernels, we again use higher-order functions because this enables the partial evaluator to optimize the constants used for them.

This higher-order function design of creating the forwarding routine would allow us to only set up the forwarding once. However, we would need to expose the returned function to C++, which is a feature that is not yet supported in AnyDSL. Thus, every time we want to denoise, we need to set up the forwarding function even though it is the same setup as before. The first three convolutions in the propagation function look like this:

```
fn @forward_denoise(img: Buffer, albedo: Buffer, normal: Buffer, mem: Buffer,
    out_buf: Buffer) -> () {
  let mut img_mat = make_matrix(img, 0, MemoryFormat::HWC, 3, height, width);
  let alb_mat = make_matrix(albedo, 0, MemoryFormat::HWC, 3, height, width);
  let nrm_mat = make_matrix(normal, 0, MemoryFormat::HWC, 3, height, width);

  let img_acc = get_mat_acc(img_mat);
  let alb_acc = get_mat_acc(alb_mat);
  let nrm_acc = get_mat_acc(nrm_mat);
```

```
    // Concatenate the 3 input images
    let cat_acc = get_cat_mat_acc(get_cat_mat_acc(img_acc, alb_acc,
        img_mat.channels), nrm_acc, img_mat.channels + alb_mat.channels);

    // Convolution 1
    img_mat = cat_conv2d(nn_int, ksize, out_channels_1, in_channels_1,
        flattened_kernels(1), biases(1), leaky_relu_x, img_mat.rows,
        img_mat.cols, cat_acc, mem, 0, size_im2col);

    // Convolution 2
    img_mat = conv2d(nn_int, ksize, out_channels_2, flattened_kernels(2),
        biases(2), leaky_relu_x, img_mat, mem, 0, size_im2col);
    img_mat = nn_int.pool(img_mat, poolsize, max_4, mem, size_im2col +
        size_img);

    let pool_1 = get_mat_acc(img_mat); // Store pool for cross-connection

    // Convolution 3
    img_mat = conv2d(nn_int, ksize, out_channels_3, flattened_kernels(3),
        biases(3), leaky_relu_x, img_mat, mem, 0, size_im2col);
    img_mat = nn_int.pool(img_mat, poolsize, max_4, mem, size_im2col + size_img
        + size_pool_1);

    let pool_2 = get_mat_acc(img_mat); // Store pool for cross-connection

    // [...] Remaining convolutions

    // Rodent expects hwc while conv2d returns chw format
    nn_int.chw_to_hwc(img_mat, out_buf, 0);
}
```

In this example, one can see that the interface is very high-level, and the programmer of the propagation function does not need much knowledge about the underlying implementation. However, the user needs to know about the memory management of the called functions since the buffers and offsets are given as an argument to every function call. As they follow a simple pattern, this should not be a huge problem.

As this already concludes the integration chapter, it shows that the application of the implementation is not complex.

5. Results

Since our goal was to implement a high-performance forwarding routine, we need some way to compare the performance our code achieves against state-of-the-art machine learning libraries. Thus, in this chapter, we present the results of some benchmarks on the different implementations and evaluate them. As preparation for this comparison, we already implemented the functionality to use BLAS libraries for the matrix multiplication in a convolution instead of ours in section 3.3.6. Applicable benchmarks are what is still missing.

For that matter, we created four benchmarks. They all use real data, dumped during the forward propagation of a sample neural network. Two of them use the denoising neural network presented in the preceding chapter, and the other two use a super-resolution neural network similar to the one Microsoft provides as an example usage for DirectML [13]. The first denoising benchmark works on a sample 512×512 image, the second on the same scene, only this time on resolution 1024×1024. Likewise, the two super-resolution benchmarks upscale once from 256×256 to 512×512 and once from 512×512 to 1024×1024. We chose to test the same networks on two different sizes to see how well they scale with the input size. In both cases, the input size is increased by a factor of 4.

When executing the benchmarks, the number of warmup and timing iterations can be set with according flags. Additionally, the benchmarks support automatic checking against a precalculated reference to ensure the calculated result was correct. Having this option can come in handy when optimizing functions internally.

We ran the described benchmarks on a system using an AMD Ryzen 5 1600X and an Nvidia Geforce GTX 1080. The results of the timing benchmarks can be seen in Table 1. On the CPU side, for each benchmark, we set the number of warmup iterations to 10 and used 100 timing iterations. On the GPU side, we performed 100 warmup iterations with 600 timing iterations. The average time it took for one iteration to complete on a given backend is given in the ‘Average’ column. The ‘im2col’ and ‘matmul’ columns describe how much time was spent performing the im2col rearrangement and the matrix multiplication, respectively. The remaining column, ‘other’, shows the fraction of time spent anywhere else.

Comparing our CPU implementation against the one using the BLAS library oneAPI, we can see that we achieve about 30% faster results across all benchmarks. Since the only difference between the two is the used matrix multiplication, one would think that all extra time is spent there. However, the im2col rearrangement is also 7% to 15% faster with our implementation. An explanation for this behavior would be that the partial evaluator can do better without the external matrix multiplication call to the BLAS API. Concerning the scaling between the different benchmarked sizes, we verify runtime scales almost proportional to the input size for both implementations. In all four cases, the factor in which runtime increased is only a fraction greater than 4.

On the GPU side, the time comparison shows that our implementation is really basic since using cuBLAS is 3 to 4 times faster overall. Unlike the implementation on the CPU, the absolute duration spent in the im2col routine is just the same here, no matter the underlying matrix multiplication. We can only observe a difference in how long multiplying the matrices takes in these cases. Regarding the scaling, our implementation took 4 times longer on the larger benchmark in the denoising case and almost 5 times longer on the super-resolution benchmark. The cuBLAS one took only between 3 and 4 times longer in both cases. This discrepancy

Benchmark	Matrix Multiplication	Time			
		Average	im2col	matmul	other
Denoise 512	AnyDSL (CPU)	463 ms	35%	64%	$< 1\%$
	oneAPI	662 ms	28%	71%	$< 1\%$
	AnyDSL (GPU)	75 ms	11%	87%	$< 1\%$
	cuBLAS	28 ms	31%	67%	$< 1\%$
Denoise 1024	AnyDSL (CPU)	1899 ms	38%	61%	$< 1\%$
	oneAPI	2794 ms	30%	69%	$< 1\%$
	AnyDSL (GPU)	304 ms	11%	88%	$< 1\%$
	cuBLAS	92 ms	38%	60%	$< 1\%$
SuperRes 512	AnyDSL (CPU)	728 ms	42%	57%	$< 1\%$
	oneAPI	1066 ms	32%	67%	$< 1\%$
	AnyDSL (GPU)	105 ms	12%	87%	$< 1\%$
	cuBLAS	31 ms	41%	58%	$< 1\%$
SuperRes 1024	AnyDSL (CPU)	3064 ms	44%	55%	$< 1\%$
	oneAPI	4271 ms	34%	65%	$< 1\%$
	AnyDSL (GPU)	509 ms	10%	89%	$< 1\%$
	cuBLAS	115 ms	44%	55%	$< 1\%$

Table 1: Timing benchmark results, using AMD Ryzen 5 1600X and Nvidia Geforce GTX 1080

is likely explained by the cache optimizations done in cuBLAS that are not considered in our implementation. A larger input might expose our already poor cache access pattern to even worse ones, as values in different rows are even further apart in memory. For cuBLAS, this problem should not occur as the cache is already taken into account by creating better layouts internally.

Across all benchmarks, on all backends, we can observe that the percentage spent in im2col increases, while the percentage spent in the matrix multiplication decreases when scaling up the input size. Consequently, we can conclude that the matrix multiplication scales better to larger inputs than the im2col routine. Nonetheless, even on the scaled-up benchmarks, most of the time is spent multiplying matrices.

Since we want a deeper understanding of the time differences we can observe in Table 1, we executed the same benchmarks again, this time measuring other, more informative metrics. On the CPU, we used the Linux performance counter tool perf [45]. We kept track of the cache and branch misses the program produced, which are arguably the most important performance measures on the CPU. Results of these measures are shown in Table 2. The third column labeled 'backend cycles idle' shows the percentage of cycles that were stalled in the backend, e.g., due to waiting for memory.

We can observe that while our implementation consistently predicts the branches better by about a factor of 2, it does a considerably worse job at utilizing cache. Interestingly, although there are much more cache misses, our implementation spends about 4 to 5 times less time waiting for memory accesses. A possible first explanation is that a non-neglectable portion of that time is due to the iteration over the result to add the bias and apply the activation function. However, even with this iteration removed entirely, the numbers stay the same. This suggests that AnyDSL, with its partial evaluation, helps a lot in making the code run efficiently.

Regarding the scaling between the input sizes, one can recognize that the input size does not matter for these measures. The only parameters influencing these metrics are the backend and the neural network architecture. This examination is consistent with the observation

Benchmark	Matrix Multiplication	perf results		
		branch misses	L1-dcache misses	backend cycles idle
Denoise 512	AnyDSL (CPU)	0.15%	26.3%	14.6%
	oneAPI	0.27%	12.4%	71.8%
Denoise 1024	AnyDSL (CPU)	0.15%	26.6%	14.8%
	oneAPI	0.26%	12.7%	68.8%
SuperRes 512	AnyDSL (CPU)	0.12%	35.1%	14.5%
	oneAPI	0.27%	21.9%	68.0%
SuperRes 1024	AnyDSL (CPU)	0.11%	35.9%	15.4%
	oneAPI	0.28%	23.3%	67.5%

Table 2: CPU performance benchmark results, using AMD Ryzen 5 1600X and Nvidia Geforce GTX 1080

that scaling the input scales every operation in the network by that same factor. Since we optimized for L1 cache usage, the input size should not affect this metric. The branch misses are also mostly independent, as scaling the input size scales both the missed branches and the correct branches by the same factor.

Since new hardware releases are far more optimized for propagating neural networks, it is interesting to run these benchmarks with the same configuration on more recent hardware. The corresponding time benchmarks in Table 3 show that our CPU implementation is consistently and significantly faster on the newer hardware too. Interestingly, our matrix multiplication performs so well that it has a timeshare of less than 50%. For the GPU, we observe that the implementation using cuBLAS scales a bit better to more recent hardware than our naïve matrix multiplication.

Table 4 shows the results of the performance benchmark. On this hardware, the branch predictor does considerably better with the library implementation than with ours. However, as the number of branch misses is still relatively low, it does not carry too much weight. Comparing this with the older hardware, we achieve fewer cache misses, which we can explain with the larger cache size of the new CPU. Nonetheless, the improvement is more noticeable in our implementation. A faster CPU needs new data earlier to continue working. Thus, we notice an increase in the idle backend cycles of the newer hardware.

5.1 Real-Time Viability

To evaluate the real-time viability of our implementation, the benchmarks in Table 1 and Table 3 are most relevant. In addition, we measured the rendered frames per second in Rodent when performing live denoising. The results are shown in Table 5. Note that when measuring FPS, the rendering time needs to be taken into account as well, and that Table 1 does, therefore, not contain the actual frame time. In the end, it is possible to use neural networks in real-time with our implementations, as long as high frames per second are not the fundamental goal. Using the most recent hardware and training smaller and thus faster networks may increase real-time viability further.

Benchmark	Matrix Multiplication	Time			
		Average	im2col	matmul	other
Denoise 512	AnyDSL (CPU)	196 ms	57%	42%	< 1%
	oneAPI	484 ms	22%	77%	< 1%
	AnyDSL (GPU)	31 ms	17%	82%	< 1%
	cuBLAS	18 ms	29%	69%	< 1%
Denoise 1024	AnyDSL (CPU)	942 ms	67%	32%	< 1%
	oneAPI	2176 ms	30%	69%	< 1%
	AnyDSL (GPU)	125 ms	16%	83%	< 1%
	cuBLAS	52 ms	39%	60%	< 1%
SuperRes 512	AnyDSL (CPU)	301 ms	64%	35%	< 1%
	oneAPI	856 ms	25%	74%	< 1%
	AnyDSL (GPU)	44 ms	18%	81%	< 1%
	cuBLAS	18 ms	43%	56%	< 1%
SuperRes 1024	AnyDSL (CPU)	1363 ms	70%	29%	< 1%
	oneAPI	3531 ms	29%	70%	< 1%
	AnyDSL (GPU)	180 ms	18%	81%	< 1%
	cuBLAS	66 ms	49%	50%	< 1%

Table 3: Timing benchmark results, using AMD Ryzen 9 3950X and Nvidia GeForce RTX 2080 SUPER

Benchmark	Matrix Multiplication	perf results		
		branch misses	L1-dcache misses	backend cycles idle
Denoise 512	AnyDSL (CPU)	0.27%	21.5%	23.9%
	oneAPI	0.16%	12.6%	75.6%
Denoise 1024	AnyDSL (CPU)	0.27%	24.2%	31.0%
	oneAPI	0.23%	14.6%	69.7%
SuperRes 512	AnyDSL (CPU)	0.34%	31.5%	24.3%
	oneAPI	0.25%	20.3%	72.2%
SuperRes 1024	AnyDSL (CPU)	0.46%	30.3%	28.7%
	oneAPI	0.34%	19.2%	70.8%

Table 4: CPU performance benchmark results, using AMD Ryzen 9 3950X and Nvidia GeForce RTX 2080 SUPER

Matrix Multiplication	FPS in 512×512	FPS in 1024×1024
AnyDSL (CPU)	1.95	0.48
oneAPI	1.40	0.33
AnyDSL (GPU)	9.9	2.5
cuBLAS	18.6	5.5
OIDN	3.34	0.96
No Denoise	51.5	14.4

Table 5: FPS when rendering the scene shown in Figure 7 with live denoising on an AMD Ryzen 5 1600X with an Nvidia Geforce GTX 1080; Rendered on the GPU

6. Conclusion and Future Work

In this thesis, we briefly summarized the concept and background of artificial neural networks. In particular, we explained how convolutional neural networks work in detail since this was the focus of this work. Furthermore, we presented a way of implementing a native CNN propagation in AnyDSL and compared its performance against implementations using state-of-the-art BLAS libraries for the matrix multiplications. The benchmarks show that it is possible to achieve comparable or even better performance, at least on the CPU side. For the implementation, we concentrated only on a subset of layers a CNN can use, e.g., we did not implement transposed convolutions. However, broadening these functionalities should not pose any problems in future work using the implemented framework.

As we mainly focused on the CPU implementation, it would be interesting to see if, with additional optimizations, a GPU implementation could compete with commonly used matrix libraries. Furthermore, since all convolutions used the im2col rearrangement, benchmarking another implementation, using the Winograd fast convolution method [29] could provide valuable results as well. Especially on small filters, as we used them, this algorithm could perform noticeably better than the im2col method.

We provide a high-level interface to access the neural network functionalities we implemented. Nonetheless, one weakness is that the programmer still needs to know about the low-level memory management of all the functions they call. An automatic memory management system in the background could solve this problem. With the limited functionality the framework provides at the moment, this might be easily integrable. However, the more different layers one integrates, the more complex this task becomes.

Acknowledgements

I would like to thank my advisor, Ömercan Yazici, for answering all my questions, providing his more recent computer for the benchmarks, and helping me debug parts of my implementation. Likewise, I would like to thank Arsène Pérard-Gayot, my former advisor, for all his insights into Rodent and the art of programming efficiently in AnyDSL. I am grateful to Prof. Dr. Philipp Slusallek and the computer graphics group for the opportunity to work in this area and on this exciting topic. I thank Prof. Dr. Sebastian Hack for agreeing to be the second reviewer for this thesis.

Additionally, I want to acknowledge Hendrik Junkawitsch, who concurrently worked on training and testing denoising neural networks, for the lively and fruitful discussions and for providing me with a neural network to integrate into Rodent.

Last but not least, I want to thank my sister Sarah-Jean Meyer for proofreading (parts) of this thesis and assisting me through my whole course of studies up to now whenever I needed help.

References

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[2] Josh Alman and Virginia Vassilevska Williams. "A refined laser method and faster matrix multiplication". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 522–539.

[3] *Basic Linear Algebra on NVIDIA GPUs*. URL: `https://developer.nvidia.com/cublas` (visited on 13/12/2021).

[4] Markus Bläser. "Fast matrix multiplication". In: *Theory of Computing* (2013), pp. 1–60.

[5] Antoni Buades, Bartomeu Coll and Jean-Michel Morel. "A review of image denoising algorithms, with a new one". In: *Multiscale modeling & simulation* 4.2 (2005), pp. 490–530.

[6] Harold C Burger, Christian J Schuler and Stefan Harmeling. "Image denoising: Can plain neural networks compete with BM3D?" In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 2392–2399.

[7] Chakravarty R Alla Chaitanya et al. "Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder". In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), pp. 1–12.

[8] Kumar Chellapilla, Sidd Puri and Patrice Simard. "High performance convolutional neural networks for document processing". In: *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft. 2006.

[9] *Clang: A C language family frontend for LLVM*. URL: `https://clang.llvm.org` (visited on 13/12/2021).

[10] Ronan Collobert, Koray Kavukcuoglu and Clément Farabet. "Implementing neural networks efficiently". In: *Neural networks: tricks of the trade*. Springer, 2012, pp. 537–557.

[11] Kostadin Dabov et al. "Image denoising with block-matching and 3D filtering". In: *Image Processing: Algorithms and Systems, Neural Networks, and Machine Learning*. Vol. 6064. International Society for Optics and Photonics. 2006, p. 606414.

[12] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[13] *DirectML*. URL: `https://github.com/microsoft/DirectML` (visited on 13/12/2021).

[14] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[15] Shi Guo et al. “Toward convolutional blind denoising of real photographs”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1712–1722.

[16] Kaiming He, Jian Sun and Xiaoou Tang. “Guided image filtering”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.6 (2012), pp. 1397–1409.

[17] David Hunter et al. “Selection of proper neural network sizes and architectures—A comparative study”. In: *IEEE Transactions on Industrial Informatics* 8.2 (2012), pp. 228–240.

[18] Roberto Ierusalimschy, Luiz Henrique De Figueiredo and Waldemar Celes Filho. “Lua—an extensible extension language”. In: *Software: Practice and Experience* 26.6 (1996), pp. 635–652.

[19] *Intel® oneAPI Math Kernel Library*. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html` (visited on 13/12/2021).

[20] *Intel® Open Image Denoise*. URL: `https://www.openimagedenoise.org` (visited on 13/12/2021).

[21] Katarzyna Janocha and Wojciech Marian Czarnecki. “On loss functions for deep neural networks in classification”. In: *arXiv preprint arXiv:1702.05659* (2017).

[22] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[23] Hendrik Junkawitsch. “Denoising Monte-Carlo Noise with Deep Learning; A Rodent-Specific Approach”. In: 2021.

[24] James T Kajiya. “The rendering equation”. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150.

[25] Martin Kolarik, Radim Burget and Kamil Riha. “Upsampling Algorithms for Autoencoder Segmentation Neural Networks: A Comparison Study”. In: *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE. 2019, pp. 1–5.

[26] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[27] Monica D Lam, Edward E Rothberg and Michael E Wolf. “The cache performance and optimizations of blocked algorithms”. In: *ACM SIGOPS Operating Systems Review* 25.Special Issue (1991), pp. 63–74.

[28] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[29] Andrew Lavin and Scott Gray. “Fast algorithms for convolutional neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4013–4021.

[30] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[31] Roland Leißa, Marcel Köster and Sebastian Hack. "A graph-based higher-order intermediate representation". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 202–212.

[32] Roland Leißa et al. "AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.OOPSLA (4–9 Nov. 2018). HiPEAC 2018 Paper Award, 119:1–119:30.

[33] Weibo Liu et al. "A survey of deep neural network architectures and their applications". In: *Neurocomputing* 234 (2017), pp. 11–26.

[34] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[35] Simon Moll and Sebastian Hack. "Partial control-flow linearization". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 543–556.

[36] Grégoire Montavon, Wojciech Samek and Klaus-Robert Müller. "Methods for interpreting and understanding deep neural networks". In: *Digital Signal Processing* 73 (2018), pp. 1–15.

[37] *NVIDIA CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit (visited on 13/12/2021).

[38] *NVIDIA cuDNN - GPU accelerated deep learning*. URL: https://developer.nvidia.com/cudnn (visited on 13/12/2021).

[39] Chigozie Nwankpa et al. "Activation functions: Comparison of trends in practice and research for deep learning". In: *arXiv preprint arXiv:1811.03378* (2018).

[40] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[41] *OpenBLAS: An optimized BLAS library*. URL: https://www.openblas.net (visited on 13/12/2021).

[42] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[43] Pawan Patidar et al. "Image de-noising by various filters for different noise". In: *International journal of computer applications* 9.4 (2010), pp. 45–50.

[44] Arsène Pérard-Gayot et al. "Rodent: Generating Renderers without Writing a Generator". In: *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2019)* 38.4 (28 July–1 Aug. 2019), 40:1–40:12.

[45] *Perf*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 13/12/2021).

[46] Olaf Ronneberger, Philipp Fischer and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[47] John E Stone, David Gohara and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.

[48] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[49] Field G. Van Zee and Robert A. van de Geijn. “BLIS: A Framework for Rapidly Instantiating BLAS Functionality”. In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), 14:1–14:33.

[50] Bing Xu et al. “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (2015).

[51] Jason Yosinski et al. “Understanding neural networks through deep visualization”. In: *arXiv preprint arXiv:1506.06579* (2015).

[52] Matthew D Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks”. In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[53] Hang Zhao et al. “Loss functions for image restoration with neural networks”. In: *IEEE Transactions on computational imaging* 3.1 (2016), pp. 47–57.