Efficient Semifield Convolutions

Peter Adema
14460165

Honours Thesis extension
Credits: 6 EC

Bachelor *Kunstmatige Intelligentie*

University of Amsterdam
Faculty of Science
Science Park 900
1098 XH Amsterdam

Supervisor
Dr. ir. R. van den Boomgaard

Informatics Institute
Faculty of Science
University of Amsterdam
Science Park 900
1098 XH Amsterdam

Semester 2, 2024-2025

Abstract TODO

Acknowledgements TODO

Contents

Chapter 1

Introduction TODO

1.1 Related work TODO

Chapter 2

Convolutional derivatives

To perform semifield convolutions within the context of a deep-learning application, we must ensure that we can take the gradient with respect to our inputs in an efficient manner for all operations we seek to perform. The primary challenge in this case is not necessarily the gradient for $\otimes$, as this is a scalar function applied once within each path in the computational graph. Instead, the challenge lies in calculating the derivative of the semifield summation $\bigoplus$, as all terms within this reduction could influence the gradient of all others. While it would be possible to simply perform backpropagation through every instance of $\oplus$ separately, this would be very computationally expensive and likely numerically unstable. As such, we would like to define semifield operators for the cases when we can find a more efficient method to calculate the derivative of $\oplus$.

We have identified three cases in which this is possible, and will discuss them in order of increasing implementation complexity. First, we will examine the possibility of using operators provided by PyTorch to implement semifield convolutions. Afterwards, we will show the two common cases for which a reduction has a simple analytical derivative, namely for those operators $\oplus$ which act like max, and those which are invertible.

2.1 PyTorch reduction operators

The most straight-forward method for implementing semifield convolutions for PyTorch models is clearly to use reduction operators provided by PyTorch itself. These operators have their derivatives provided by PyTorch, and are integrated into the PyTorch automatic differentiation system such that writing a backward pass is not required.

The challenge when using PyTorch operators lies then not in their derivative, but in how to apply them in the correct fashion: the axes we would wish to reduce over are not the image axes present in the input itself, but instead the window axes of every sliding window we extract during the convolution.

A built-in method for accomplishing this is `nn.Unfold`, an operator which examines sliding windows from the input and copies each separately to a new buffer, allowing subsequent operators to work with the window axes of the new buffer. This copying is rather suboptimal, however: if we take a relatively small 32×32 image and extract relatively small 5×5 windows from it, then the result is more than 19 times the size of the image: even ignoring the cost of copying, larger inputs or kernel sizes would quickly lead to memory issues.

As such, we would wish to perform a similar operation (transforming the input image such that we have access to sliding window axes to reduce over), but without performing a copy. This is possible by creating an adjusted view of the input image: some other libraries include this feature by default, but it is not exceedingly complex to replicate.

The main idea of using such a view is to adjust the strides (step sizes within the underlying data array) such that, when moving to the next window, the view takes a step equal to the convolutional STRIDE instead of the full size of the window. In Fig. 2.1, we can see how a view into a 1D-array with STRIDE = 1 and size-4 kernels causes subsequent windows to overlap:

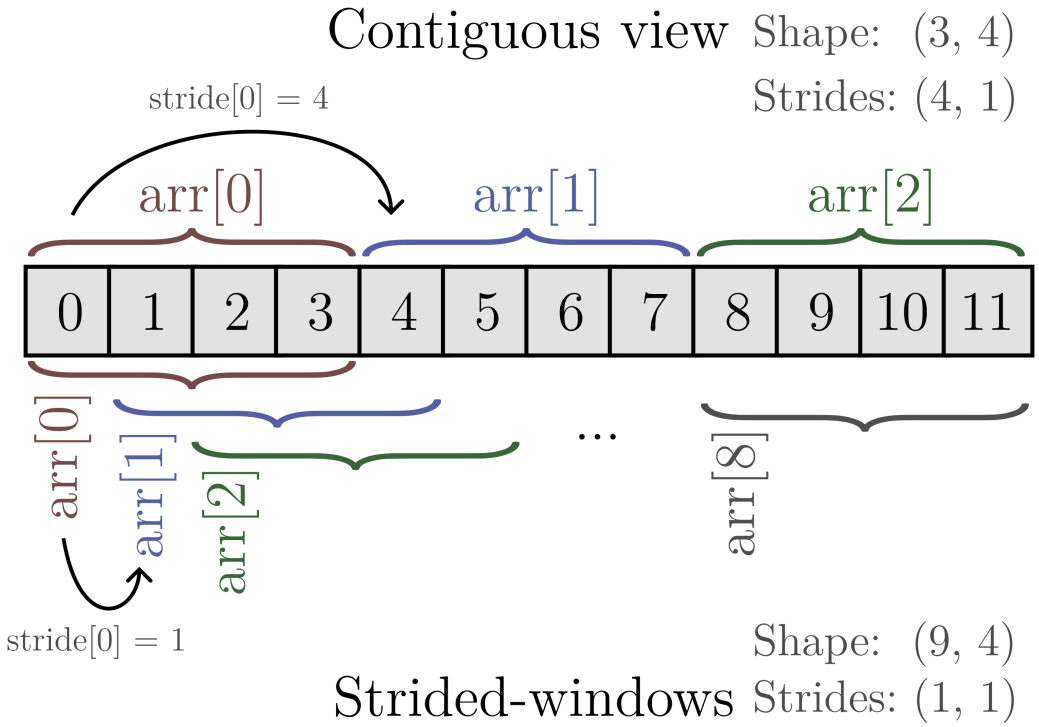

Figure 2.1: Two views onto the same underlying data: at the top, a contiguous view (the typical view for PyTorch arrays), and at the bottom, a strided-windows view (allowing for extracting windows without copying).

By applying a broadcasting version of $\otimes$ to the strided view and the convolutional kernels, we can obtain the intermediaries to be reduced. By applying the reduction operator along the window axis (axis 1 in Fig. 2.1), we can then efficiently calculate $\bigoplus$ for all cases where $\bigoplus$ and $\otimes$ can be expressed as one or more PyTorch operators.

However, this approach still has a major downside in terms of memory usage and cache locality: there is currently no method for fusing the $\otimes$ and $\bigoplus$ operations in PyTorch, and the intermediary results of $\otimes$ must therefore be manifested in their entirety. As a result, this method still suffers from greatly increased memory use but (more importantly for the small-scale experiments in the main report) also accesses memory inefficiently. Since all results of $\otimes$ are stored an calculated before $\bigoplus$ begins, the initial results of $\otimes$ are flushed from the cache by the beginning of $\bigoplus$, causing further slowdowns from cache-misses.

In order to mitigate this, we must devise custom, fused kernels that combine $\otimes$ and $\bigoplus$ into a single operation. This, however, also requires calculating the derivative rules manually; the next two sections will discuss two such rulesets.

2.2 Selection-based semifield addition TODO

If it acts like max, we can memoize the provenance (which value was selected).

2.3 Invertible semifield addition TODO

If we can undo it, we can use [1].

Chapter 3

CUDA semifield convolutions TODO

Armed with an understanding of the types of semifield convolutions where a gradient can be calculated in a reasonably efficient manner, we now turn to the task of efficiently implementing these operations as programs that can run on a (NVIDIA) GPU: CUDA kernels.

Chapter 4

PyTorch C++ Extensions
TODO

Now that we have working implementations of semifield convolutions in the form of CUDA kernels, it is important to examine how these kernels can best be used within the context of a deep-learning model created with the PyTorch machine learning framework.

Chapter 5

Conclusions TODO

5.1 Findings TODO

5.2 Discussion TODO

5.3 Contributions TODO

5.4 Further research TODO

5.5 Reproducibility TODO

5.6 Ethics Maybe?

.....

Bibliography

[1] A. Paszke, M. J. Johnson, R. Frostig, and D. Maclaurin, "Parallelism-preserving automatic differentiation for second-order array languages," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, 2021, pp. 13–23.

Chapter 6

Appendix