U-net CNN in APL

Exploring zero-framework, zero-library machine learning

Aaron W. Hsu
aaron@dyalog.com
Researcher
Dyalog, Ltd.
Bloomington, IN, United States

Rodrigo Girão Serrão
rodrigo@dyalog.com
Consultant
Dyalog, Ltd.
Bramley, United Kingdom

Abstract

The APL notation would appear to be a clear match for convolutional neural networks, but traditional implementations of APL have lagged behind the performance of highly tuned, specialized frameworks designed to execute CNNs on the GPU. Moreover, most demonstrations of APL for neural networking have involved relatively small examples. We explore a more complex example in the U-net architecture and utilize a modern APL compiler with GPU support, Co-dfns, to compare the state of the art of APL against the current crop of specialized neural network frameworks in the form of PyTorch. We compare performance as well as the language design of APL for neural network programming and the clarity and transparency of the resulting code.

We found that the complete "from scratch" APL source was on par with the complexity of the PyTorch reference implementation, albeit more foreign, while being more concise and complete. We also found that when compiled with Co-dfns, despite the naive implementation both of Co-dfns and our own code, performance on the GPU and the CPU were within a factor of 2.2 - 2.4 times that of the PyTorch implementation. We believe this suggests significant avenues of future exploration for machine learning language design, pedagogy, and implementation, both inside and outside of the APL community.

ACM Reference Format:

Aaron W. Hsu and Rodrigo Girão Serrão. 2022. U-net CNN in APL: Exploring zero-framework, zero-library machine learning. In *Proceedings of International Conference on Functional Programming (ICFP'22).* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICFP'22, Sep 11 - Sep 16, 2022, Ljubljana, Slovenia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Specialized machine learning frameworks dominate the present industrial and educational spaces for deep learning applications. A wide number of higly specialized and highly optimized libraries exist, often built on top of one another, to support the modern wave of machine learning architectures. These systems are often more complex than your typical library, and they might even be better classified as their own domain-specific languages (DSLs). While these libraries have supported the current explosion of machine learning developers, a number of issues have emerged.

First, because of their highly specialized nature, users of these systems tend to become experts not in generalized programming or algorithmic skills, but specialist toolkits and frameworks around a very specific model of computation. This specialized nature often mandates dedicated courses and even entire academic specializations (even at the undergraduate level) focused on the mastery of these particular concepts. This can create a sharp fall off of skills transferrence, where machine learning experts can use machine learning frameworks effectively, but may be underdeveloped and underprepared to handle situations that require a broader or more adaptive skillset.[1]

Second, from a pedagogical perspective, when teaching machine learning, one may often be able to implement simple networks in a general-purpose programming language, but trying to teach machine learning through a typical general purpose language can be difficult, because one quickly encounters performance and scalability limitations that make any non-trivial and interesting applications likely beyond the competency and endurance of your typical student. This creates a sharp contrast in which one begins with simple systems that can be programmed "by hand" but quickly transitions to highly opaque and complex frameworks that are difficult to understand, modify, or intuit. This can result in significant reductions in professional competency.

Third, if a lack of profound and intuitive understanding of the underlying mechanics of a deep learning system continues into professional life, the result can be a type of "programming by knob turning" in which neural networks are programmed via trial and error rather than through intentional design. Machine Learning as a discipline is already

[1] To someone who only has a hammer, everything looks like a nail.

opaque enough, with many cases of unintended consequences, without the added dangers inherent in this sort of unintentional programming guesswork [Domingos 2012].

Fourth, the specificity of machine learning frameworks can result in significant amounts of code churn and a reduction in the stability of codebases for enterprise use. Switching hardware, architectures, operating systems, or the like can create unstable conditions in which code must be rewritten, adapted, or thrown away entirely. Machine learning frameworks are often highly vendor-specific, and even those which are more vendor-neutral tend to encode significantly greater specificity than is historically warranted for code intended to last for any long period of time. This almost necessitates higher levels of programmer investment in order to keep such systems running over a long period of time.

Despite the above potential issues, specialist frameworks have proven highly effective, in large part because of how important high-performance is to the domain of machine learning. However, in recent years, general-purpose array programming languages have seen a resurgence, and naturally, they have been examined in the light of machine learning. Such languages were also popular during early exploration of neural network programming during the 20th century [Alfonseca 1990], but performance issues of then-current hardware prevented further progression.

APL, as a general-purpose array programming language, created by Kenneth Iverson as an improved mathematical notation [Iverson 1962], has seen an increase in popularity over the past decades, in part because of the renewed interest in parallel computation and a wider acceptance of the use of a variety of programming languages. However, only recently has significant new research into the use of APL as a possible implementation language for machine learning begun to surface.

The long history of APL, its origins as an pedagogical tool, and its reputation for directness of algorithmic expression [Knuth 1993, 2007] help to address some of the concerns above. Furthermore, it is one of the most linguistically stable languages, while also being exceptionally high level and high performance at the same time [Hsu 2019], making it highly suitable for long lived code as well as rapid prototyping. Finally, the language itself defaults to a data-parallel semantics, making its application to GPU programming an obvious conclusion.

While the above advantages might suggest APL as a terrific tool for machine learning, unfortunately, the vast majority of implementations have been for the CPU only, and those have usually been entirely interpreted. Traditionally, compiler implementors have considered APL a challenging language to compile [Hsu 2019], but recent innovations to the language (particularly those with a functional programming focus) have made compilation much more tractable, and the Co-dfns compiler now exists as an APL implementation with native GPU support [Hsu 2019].

Given the available APL technology and the parsity of existing materials on modern machine learning development in APL, we conducted an exploration into the state of the art in APL, both from a language design and a runtime implementation perspective. To do this, we focused our efforts on the implementation and benchmarking of the U-net convolutional neural network [Ronneberger et al. 2015]. This is a popular image segmentation architecture with a particularly interesting U-shaped design. It makes use of a range of popular CNN vocabularies and functions while having a clear architecture that is not so simple as to be trivial. This makes it an ideal candidate for exploring APL's capabilities.

We make the following contributions:

- A complete demonstration in APL of the popular U-net convolutional neural network, which is non-trivial in vocabulary and architecture
- Our U-net implementation is exceptionally simple, concise, transparent, and direct
- Our implementation was written with pure APL and no dependencies, frameworks, libraries, or other supporting code outside of the APL implementation
- A functional programming-friendly approach to neural network design and implementation
- An analysis and examination of the current language features within APL that appear relevant to CNNs and machine learning
- A critical discussion and design comparison of two different approaches to supporting convolutions and similar operations in a general-purpose array language with a recommendation for future implementation improvements
- A grounded perspective on the applications of general-purpose array programming languages like APL to the machine learning space from professional and pedagogical angles and how APL compares to alternative, specialist framework approaches
- Performance results of two modern APL implementations, one compiled and the other interpreted, on CPU and GPU hardware against a reference PyTorch implementation for U-net
- Performance observations of specialized neural network functionality exposed in more general purpose array frameworks for GPU programming
- Specific highlighting of low-hanging fruit for improving the current range of APL implementations both in terms of language design and runtime implementation
- A demonstration of the expressiveness and performance that careful language design can enable without the need for complex implementation models or theory

2 Background

In this section, we provide the relevant background pertaining to the machine learning concepts needed to work with CNNs, and the u-net in particular, and to the APL language.

2.1 Convolutional Neural Networks

The experiment this paper uses to produce its benchmarks is the reproduction of a well-known convolutional neural network architecture. The use of CNNs in machine learning was widely popularised with the publication of a paper [Krizhevsky et al. 2012] that used CNNs to achieve state-of-the-art performance in labeling pictures of the ImageNet [Deng et al. 2009] challenge. However, a proeminent paper from 1998 [LeCun et al. 1998] shows that the modern use of CNNs can be dated farther back.

The use of convolutional neural networks, as we know them today, builds on top of the convolutional layer [O'Shea and Nash 2015]. Convolutional layers receive three-dimensional tensors as input and produce three-dimensional tensors as output. These inputs have a fixed number of channels[2] n_{in} which are then transformed into n_{out} channels through means of discrete convolutions with a total of $n_{in} \times n_{out}$ kernels, the learnable parameters of the convolutional layer. One of the advantages of CNNs is that, although the total number of kernels $n_{in} \times n_{out}$ depends on the number of input and output channels, the sizes of the kernels are independent of the size of the other two dimensions of the inputs. Despite the fact that the main dynamics of a convolutional layer is governed by discrete convolution with the learnable kernels, the exact behaviour of a convolutional layer depends on layer parameters like the padding and the stride used [Dumoulin and Visin 2016].

Given that CNNs were primarily used in image recognition-related tasks, convolutional layers were often paired with pooling layers that ease the recognition of features over small neighbourhoods [Scherer et al. 2010]. The rationale behind these pooling layers, as seen from an image recognition-related context, can be interpreted as follows: the image features one is typically interested in (e.g., the recognition or segmentation of objects, or image labeling) are not contained in single pixels of the input images, but in regions of said pixels. Pooling layers are, thus, employed with the purpose of aggregating low-level information that can then be used to recognise the larger features of interest [Scherer et al. 2010].

In 2015, three authors published a paper [Ronneberger et al. 2015] introducing the u-net architecture: a CNN with a non-trivial architecture that won several biomedical image segmentation challenges at the time of its publication. Since then, the u-net architecture was reimplemented hundreds of times[3], most notably through the use of deep-learning frameworks such as PyTorch [Paszke et al. 2019], a deep-learning framework used in this work, or Caffe [Jia et al. 2014], which is the deep learning framework in which the original u-net was implemented. For this paper, we reimplemented the u-net architecture, in APL, without making use of any (machine learning) libraries or frameworks. Before we introduce our work on that implementation, we discuss the original architecture that we set out to replicate.

2.2 Original U-net Architecture

Figure 1 shows the original diagram that represents the u-net architecture [Ronneberger et al. 2015], which we cover now. We will go through the figure from left to right, following the U-shape of the diagram.

The blue right arrows, labeled "conv 3x3, ReLU", represent unpadded convolutions with 3×3 kernels. Figure 1 shows that after each of these convolutions, the size of the feature maps decreases by 2, from which it can be inferred that the stride [Dumoulin and Visin 2016] is 1. After each convolution, we use the activation function rectified linear unit (ReLU) [Nwankpa et al. 2018]. Pairs of these convolutions are followed by max-pooling operations represented by the red down arrows. These max-pooling operations act on a 2×2 region and have a stride of 2, effectively downsampling each feature map to half the size. Because of this repeated halving, the input size must be chosen carefully[4]. After every downsampling step, the first convolution doubles the number of channels. The pattern of two convolutions (with ReLUs) followed by downsampling via max-pooling happens four times and makes up the contracting path of the network, on the left of Figure 1.

Having reached the end of the contracting path (at the bottom of the diagram), we start the expanding path. The expanding path also makes use of unpadded convolutions with 3×3 kernels and stride 1, but these are now at the end of each step, instead of at the beginning. Each step of the expanding path starts with an upsampling operation (green up arrows) that doubles the size of the feature maps while cutting their number down in half. For this upsampling, we infer that the original authors used a transposed convolution (with 2×2 kernels) of stride 2 [Dumoulin and Visin 2016][5]. These transpose convolutions produce half of the channels that are fed as input to the regular convolutions. The other half of the channels is copied and cropped from the corresponding step in the contracting path, as represented by the gray right long arrows in the middle of the diagram of Figure 1. Because there is a slight mismatch between the size of the feature maps that are copied and cropped and the other feature maps that resulted from the upsampling step, the

[2]"channel" typically refers to the leading dimension of these inputs/outputs, a nomenclature that is derived from the fact that CNNs were popularised in the context of image processing.

[3]Numbers by Papers with Code as of March, 2022.

[4]Specifically, the input dimensions must be congruent to $12 \mod 16$

[5]See [Serrão 2022] for an informal discussion of this inferrence.

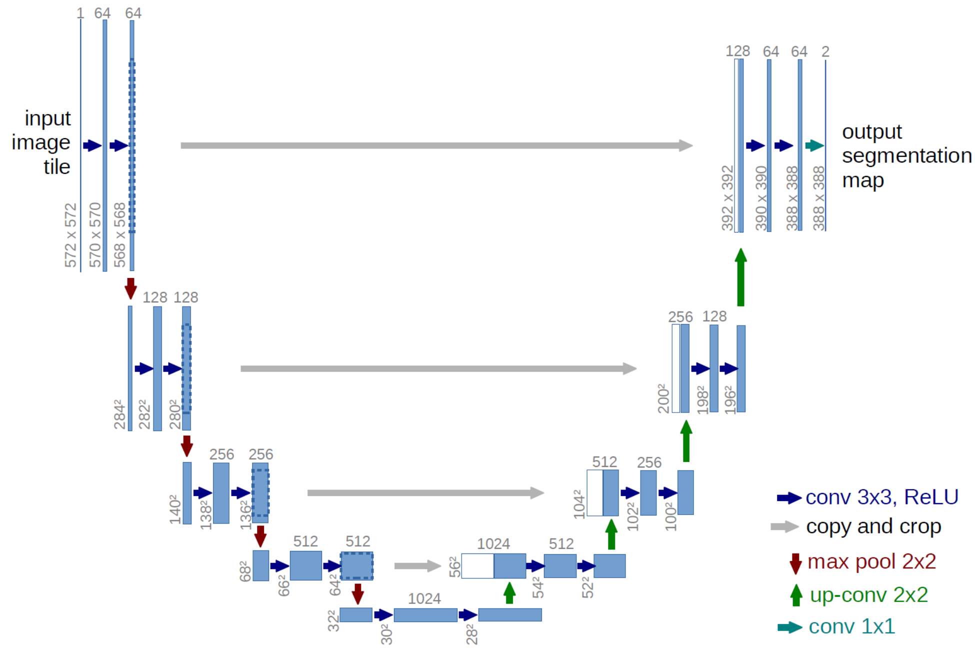

Figure 1. Original u-net architecture, as seen in the original paper [Ronneberger et al. 2015]. Arrows represent operations between the multi-channel feature maps represented by the rectangles. The number on top of each rectangle is its number of channels and the numbers in the lower-left corner are the x and y dimensions of the feature maps.

feature maps that are copied get cropped from the centre of the larger feature maps of the contracting path. It is after this copy and crop operation that we feed the feature maps into the two convolution layers that are paired with their respective ReLU activation functions.

At the end of the contracting path, we have a 1×1 unpadded convolution that reduces the 64 feature maps to 2 feature maps (one per class).

To compute the loss of the output with respect to the expected labels, we compute the softmax across the two output channels followed by the cross entropy loss function.

2.3 APL Notation

APL [Iverson 1962] is an alternative mathematical notation, introduced by Turing award winner Kenneth E. Iverson in the '60s, that has since evolved into an executable mathematical notation [Hui and Kromberg 2020]. In this section, we introduce the basics of the APL notation, but the reader is directed to [Legrand 2009] for a full tutorial. Online interactive systems are also available[6], which should make it easier to get acquainted with APL. Throughout the remainder of this paper, the notation used is such that it is compatible with Dyalog APL 18.0[7].

[6]TryAPL https://tryapl.org is an example of such a service.

[7]You can get Dyalog APL from Dyalog's website https://dyalog.com/download-zone.htm.

2.3.1 Functions and Arrays.

APL is an "alternative" mathematical notation because it differs from the traditional mathematical notation in some ways. However, not everything in APL is foreign, as demonstrated by the following examples of addition and multiplication:

```
      1 + 2
3
      73 × 104
7592
```

The format of the two examples above will be the same throughout the paper[8]: the notation typed by the user is indentend to the right and the computed result is left-aligned on the following line(s). Subtraction and division are also represented by the usual glyphs, `-` and `÷`, respectively:

```
      10 - 1 2 3
9 8 7
      100 50 20 ÷ 2
50 25 10
```

In APL, one is allowed to write multiple values next to each other, which are then *stranded* together and interpreted as a vector. Thus, `1 2 3` represents the three-item vector whose elements are the first three positive integers. Then, the APL function *minus* takes the scalar `10` as its left argument and the three-item vector `1 2 3` as its right argument, and it subtracts each of the items of the right argument vector

[8]This format mimics that of the APL session, the interactive environment in which one can use APL.

from its left argument. Similarly, the division example shows that vectors can also be used as the left argument. The natural progression is to wonder whether vectors can be used on the left and on the right of a function, and typically they can. We demonstrate that with the max function, represented by the upstile glyph `⌈`:

```
      (1⌈5) (10⌈5) (100⌈500) (1000⌈500)
5 10 500 1000
      1 10 100 1000 ⌈ 5 5 500 500
5 10 500 1000
```

The first example shows how parenthesis `()` can be used to create vectors whose items are results of other expressions, given that the four expressions inside parenthesis produced the four items of the result vector. The second example shows that we can obtain the same result by collecting all the left arguments inside all `()` on the left of a single `⌈`, and by collecting all the right arguments inside all `()` on the right of that same `⌈`.

The dyadic functions *plus* `+`, *minus* `-`, *times* `×`, *divide* `÷`, and *max* `⌈`, all share the property that allows them to accept vectors as arguments: they are *scalar functions*. Scalar functions are functions that pervade the structure of the argument(s) and apply directly to each of the scalars that make up said argument(s). This becomes increasingly relevant when one understands that APL has first-class support for arrays of any dimension, of which we have seen *scalars* such as `10` and `73` and *vectors*. Scalars and vectors can be typed directly but arrays of higher dimensions must be loaded from an external data source or created dynamically through computations.

The reshape function is represented by the Greek letter rho `⍴` and is a dyadic function that reshapes its right argument to have the shape specified by the left argument. For instance, if we want to create a 2×3 matrix with the first six non-negative integers, we can do it like so:

```
      2 3 ⍴ 0 1 2 3 4 5
0 1 2
3 4 5
```

Each non-negative integer of the left argument specifies the length of the result along the corresponding dimension. So, if the left argument had been `5 9 7`, the resulting array would have been a cuboid (array with three dimensions) composed of `5` planes, `9` rows, and `7` columns, holding a total of $5 \times 9 \times 7 = 315$ items.

Given an arbitrary array `array`, we can also use the Greek letter rho `⍴` to compute the *shape* of the array, that is, the length of each of its *axis*, or dimensions. In the example below, we can see that `array` is a matrix with `2` rows and `3` columns, even though we don't know what the items of `array` are:

```
      ⍴array
2 3
```

This also goes to show that many functions have two behaviours, one monadic behaviour and one dyadic behaviour. A function is used monadically when it has an array argument on its right, but not on its left, and a function is used dyadically when it has an array argument on its left and another one on its right. For example, rho `⍴` represents the monadic function *shape* and the dyadic function *reshape*. In this particular instance, we can also see that `array` is a *nested* matrix:

```
      array
┌───┬───┬───┐
│0 0│0 1│0 2│
├───┼───┼───┤
│1 0│1 1│1 2│
└───┴───┴───┘
```

The cells above each contain a two-item vector (`0 0` through `1 2`), and the borders surrounding those two-item vectors are a visual cue to help the reader discern the nested nature of the array.

Another key difference between the APL notation and the traditional mathematical notation is that APL normalises precedence rules by saying that all functions have the same precedence: functions are said to have a *long right scope* and a *short left scope*, which is why APL is often said to "execute from right to left". A long right scope means that a function takes as right argument everything to its right, whereas a short left scope means that a function only takes as left argument the array that is immediately to its left. The expression $2 \times 3 - 4 \times 5$, in standard mathematical notation, is equivalent to $(2 \times 3) - (4 \times 5) = 6 - 20 = -14$, because multiplication has higher precedence over subtraction. However, the APL expression `2×3-4×5` is equivalent to `2×(3-(4×5))`:

```
      (2 × 3) - (4 × 5)
¯14
      2 × (3 - (4 × 5))
¯34
      2 × 3 - 4 × 5
¯34
```

APL uses the high-minus `¯` to represent negative numbers, otherwise there would be ambiguity in the use of the minus sign `-`[9].

2.3.2 Shape, Rank, Data.

Every APL array can be fundamentally characterised by its *shape*, its *rank*, and its data:

- the *shape* of an array can be computed with the function *shape* and is a vector that specifies the length of each dimension of its argument;
- the *rank* of an array is the number of its dimensions (the length of its *shape*) and dictates the name of said array as per Table 1; and

[9] Is `1 -2` the APL expression "one minus two" or the two-item vector "one, negative two"?

Rank	*Name*
0	scalar
1	vector
2	matrix
3	cuboid

Table 1. Array names according to *rank*.

- the data of an array are the items that compose said array.

The *shape* of an array `arr` is `⍴arr`. The *rank* of an array is the *length* of its *shape* or, in APL vocabulary, the *tally* of its *shape*, which is `≢⍴`. Finally, the data of an array can be retrieved as a vector with the monadic function *ravel* `,` (comma). These are illustrated below for a matrix. We use the primitive *roll* `?` to fill the matrix with random data and annotate the expressions with APL comments `⍝`:

```
      ⍝ Create random array mat
      mat ← ?2 3⍴0
      mat
0.999 0.00424 0.351
0.967 0.92    0.821
      ⍝ mat has shape 2 3.
      ⍴mat
2 3
      ⍝ mat has rank (tally shape) 2
      ≢⍴mat
2
      ⍝ mat contains this data:
      ,mat
0.999 0.00424 0.351 0.967 0.92 0.821
```

2.3.3 Operators.

On top of providing a rich set of built-in functions, APL provides a series of operators that allow us to combine and modify our functions. A typical example of a monadic APL operator is *reduce-first* `⌿`. The monadic operator *reduce-first* takes a function on its left and then inserts it between the elements of the right argument. Previously, we computed the total number of elements in a cuboid with shape `5 9 7` by inserting the *times* function between each pair of numbers. With *reduce-first*, this can be simplified:

```
      5×9×7
315
      ×⌿5 9 7
315
```

The function *times*, together with the operator *reduce-first*, creates the *derived function* `×⌿`, recognised as the function *product*. Similarly, the derived function `+⌿` is the function *sum*:

```
      1+2+3+4
10
      +⌿1 2 3 4
10
```

This highlights the versatility of APL in that operators combine with a variety of functions. Another source of versatility in APL comes from how functions get applied to arrays of different ranks.

The operator *reduce-first* `⌿` gets its name by contrast with the operator *reduce* `/`, given that the two operators differ in the axis along which their derived functions operate. With the help of the function *index generator* `⍳` and the *left arrow* `←` that performs assignment, we can create a matrix `mat` with shape `2 3` and demonstrate the difference between the two derived functions `+/` and `+⌿`:

```
      mat ← 2 3⍴⍳6
      mat
0 1 2
3 4 5
      +/mat
3 12
      +⌿mat
3 5 7
```

Plus-reduce-first `+⌿` sums along the first axis of its argument and *plus-reduce* `+/` sums along the last axis of its argument. For higher-rank arrays, an arbitrary axis can be specified with the *axis operator* `[axis]`. For example, the construct `+/[0]` uses *reduce* with *axis* to replicate the behaviour of `+⌿`.

On top of monadic operators, that take a single operand on the left, APL provides a series of dyadic operators that take a left operand and a right operand. One such dyadic operator is the *inner product* `.` (dot), which we use thoroughly for the derived function *matrix product* `+.×`[10]. We exemplify *matrix product* below:

```
      X ← 2 3⍴0 0 0 1 10 100
      Y ← 3 2⍴1 2 3 4 5 6
      X Y
┌──────────┬───┐
│0  0     0│1 2│
│1 10   100│3 4│
│          │5 6│
└──────────┴───┘
      X +.× Y
  0   0
531 642
      Y +.× X
2 20 200
4 40 400
6 60 600
```

APL functions can only take arrays as arguments, but APL operators can take functions or arrays as operands. The operator *rank* `⍤` is one such operator, which takes the forms `X`

[10] Presenting the operator *inner product* in all its generality is outside the scope of this paper.

`(f⍤A) Y` and `(f⍤A) Y`, where `X` and `Y` are arbitrary arrays, `A` is a scalar or a one-item vector (or a two-item vector if `X` is present), and `f` is a function. The derived function is such that, instead of operating on the full argument(s), operates on subarrays of the specified rank(s) specified in `A`:

```
      mat ← 2 4⍴⍳8
      mat
0 1 2 3
4 5 6 7
      mat (×⍤1 0) 1 ¯1
 0  1  2  3
¯4 ¯5 ¯6 ¯7
```

The matrix `mat` has two subarrays of rank one, its rows; and the vector `1 ¯1` has two subarrays of rank zero, its items, thus `(×⍤1 0)` will multiply the rows of `mat` with the items of `1 ¯1`, resulting in a matrix that has the same first row and a negated second row as `mat`.

2.3.4 User-defined Functions and Operators. In APL, one can use the *direct functions (dfns)* syntax to create user-defined functions, which can then be named and reused throughout the APL programs. A dfn is enclosed by braces `{}` and it can only take a right argument, or a left and right argument. Inside a dfn, we use *omega* `⍵` to refer to the right argument and *alpha* `⍺` to refer to the left argument. We provide a short example:

```
      vec ← 0 1 2 3
      ⍝ Sum of vec divided by its tally
      (+⌿vec)÷≢vec
1.5
      ⍝ Sum of arg divided by its tally
      avg ← {(+⌿⍵)÷≢⍵}
      avg vec
1.5
```

Similarly, the *direct operators (dops)* syntax can be used to create user-defined operators. A dop is also enclosed by braces `{}` and it can only take a left operand `⍺⍺` if it is a monadic operator, or a left `⍺⍺` and a right `⍵⍵` operand if it is a dyadic operator. Inside a dop, `⍵` and `⍺` still refer to the arguments of the derived function. For example, given a dyadic function `f` and a monadic function `g`, the pattern `(g X) f g Y`[11] can be abstracted away with the dop `{(⍵⍵ ⍺) ⍺⍺ ⍵⍵ ⍵}`[12].

3 Implementation

3.1 Overview

Our implementation of u-net can be roughly divided into two significant considerations: the implementation of the fundamental vocabulary of neural networks, and the wiring of those operations into the actual u-net architecture. We leveraged significant features of APL to implement both aspects of the system, and so we will treat each in their own sub-section.

Additionally, because Co-dfns does not yet support the complete array of Dyalog primitives and their semantics, some of the implementation techniques that we use could be significantly enhanced through the use of a more rich feature-set. The effect of using these richer features is an increase in concision and clarity, but we expect that such improvements would not significantly affect the overall performance of the code, either positively or negatively. We believe that the overall structure of the code is clear and simple enough at the current state of Co-dfns to warrant inclusion almost verbatim here, rather than use the richer features and require the reader to translate those into the Co-dfns supported feature set in order to execute them.

One area that deserves particular attention is the design of APL as a language itself and the specific features that immediately present themselves as particularly well-suited to expressing neural network computations. Our exploration of these features uncovered a particular design tension that is worth discussing in detail. A complete copy of the code discussed in this paper is included in the appendix.

3.2 Design of APL Primitives for Neural Networks

The majority of APL primitives find fundamental use in computing neural networks, which isn't surprising given the array-oriented and numerical nature of the domain. However, the stencil operator, introduced in Dyalog APL [Hui 2017], stands out as the most obviously aligned with convolutional neural networks. The J programming language introduced an alternative implementation of the stencil operator earlier [JSoftware 2014], from which Dyalog derived inspiration for the implementation of their own stencil operator.

The stencil operator takes a function left operand and a specification array as a right operand. Given a function f as the left operand and a specification s as the right operand, the stencil operator, written f⌺s, evaluates to a function that applies f on each sliding window specified by s. The two most common sliding window sizes for stencil in u-net are $3\;3$ for the convolutions, corresponding to a window size of 3×3 and a step of 1 for each dimension, and $2\;2\rho 2$ for the max pooling layers and up convolutions, corresponding to a 2×2 window size and a step of 2 for each dimension.

When first implementing a convolution, almost everyone familiar with Dyalog APL and the stencil operator immediately comes to some variation of the following expression for convolving a matrix M with a kernel K:

$$\{+/,K \times \omega\}\text{⌺}3\;3 \vdash M \tag{1}$$

Recall that $K \times \omega$ is the pointwise multiplication of kernel K with one of the 3×3 sliding windows. We write $+/,A$

[11] A helpful interpretation of this pattern is "preprocess the arguments to `f` with the function `g`".

[12] This is a partial model of the operator *over* `⍥` from APL.

to indicate the sum of all elements of array A. Thus, the above stencil computes the 2-D convolution over a matrix with a given 2-D kernel. Because APL's stencil operator is *leading axis biased*, if we were to instead provide a kernel K with shape $3\,3\,C$ where C is the number of channels in an array A of shape $M\,N\,C$, the above expression would still function appropriately. However, if we wish to continue to extend this to multiple kernels, that is, multiple output channels, it is less straightforward to compute. The following expression computes the convolution of an array A with shape $M\,N\,I$ using kernels K with the shape $O\,3\,3\,I$ where I is the number of input channels and O the number of output channels:

$$K\{k \leftarrow \alpha \diamond \{+/, k \times \omega\}\boxed{\diamond}3\,3 \vdash \omega\}\ddot{\circ}3 \vdash A \tag{2}$$

The result of the above expression is an array of shape $O\,M\,N$. We make use here of the rank operator, seen in 2.3.3. The expression $K\ f\ddot{\circ}3\ \vdash\ A$ will divide K and A into subarrays each of rank 3, that is, each with 3 dimensions, and apply f to the corresponding subarrays of K and A. Thus, in our expression above, our convolution will be applied over the entire A for each output channel described by the first axis of K, thus applying the original 2-D convolution over arbitrary numbers of input channels and output channels.

Unfortunately, the above expression has a number of design flaws. Firstly, the output has the channel count as the leading axis, while the expected input is to have the channel count as the trailing axis. This requires that we perform a transposition of these dimensions after computing the convolution in order to return our result to the input format. Furthermore, the nested structure of the computation results in two primary functions of non-primitive complexity, meaning that a more sophisticated analysis of this function would be required by a compile-time or run-time implementation in order to recognize this code.

On principle, APL is at its best when it can concisely describe operations over large arrays at a time, or large subarrays at a time. In particular, concise APL expressions are possible when the solution can be expressed as a composition of basic APL primitives. However, in the case of the stencil operator, almost all interesting use cases of the function come from complex, non-simple, non-primitive left operands. This is further exacerbated by the need to nest the stencil operation in an outer rank as above. Additionally, the input sizes provided to the left operand of the stencil operator are remarkably small, all things considered. This guarantees that a naive implementation of stencil will be inefficient and slow, especially on interpreters.

Dyalog APL can mitigate some of these issues through the use of idiom recognition. However, we argue that idiom recognition scales particularly poorly to this case. Idiom recognition has been implemented for the stencil operator, and there are a selected number of left operand inputs that are treated specially, so that their performance can be enhanced behind the scenes [Hui 2020]. However, because of the complex nature of the left operand inputs, recognizing the useful idioms for the stencil operator is a particularly difficult task, and does not scale well to these sorts of problems. For instance, while the above stencil operator is considered an idiom, the following is not:

$$\{\lceil\text{⌿}\lceil\text{⌿}\omega\}\boxed{\diamond}(2\ 2\rho 2) \vdash A \tag{3}$$

This expression is one way to implement part of the max pooling layers in u-net. However, there are many other obvious variations of this expression that would also have to be considered for idiom recognition, which would otherwise be missed even if this particular expression were handled. In the case of the design of the stencil operator, trying to improve its performance via idiom recognition or even compiler optimization is a relatively significant task. It is combinatorial and not compositional.

The result is that significant amounts of code would have to be implemented and maintained in order to support performance enhancements on the stencil operator, with none of that work benefiting other parts of an APL runtime.

We instead propose an alternative that was first suggested to us by the late Roger Hui, the stencil *function* [Hui 2020]. The stencil function is a function whose left argument is the same as the right operand of the stencil operator, and which receives the same right argument as the right argument to the function returned by the stencil operator. A reasonable definition of the stencil function might be:

$$SF \leftarrow \{\{\omega\}\boxed{\diamond}\alpha \vdash \omega\} \tag{4}$$

We found that using the stencil function was in fact, not only easier to work with and more compositional than the stencil operator, but that it was also universally faster. That is, even with the idiom recognition that Dyalog has put into their interpreter to handle the special cases of the stencil operator, of which SF is one [Hui 2020], using the stencil function instead of the stencil operator was always at least as fast or faster, despite idiom recognition for the more complex uses of the stencil operator.

The compositionality of SF has performance ramifications, since it is fundamentally an indexing operation, rather than a computational operation. This categorical shift means that it can now be approached using the same sorts of lazy indexing and fusion operations that are common for other indexing operations, such as transposition. This means that the use of the stencil function can help to broadly reduce intermediate array generation, and performance enhancements to indexing will compose well with SF. Functions and operators that are already designed to fuse with lazily indexing functions can then readily take advantage of such features to work with the output of SF as well, granting performance enhancements across a wider range of applications without ever implementing any idiom recognition, which reduces the amount of specialized code that needs to exist as well as the programmer burden to maintain such code.

To explore this further, a naive implementation of the stencil function that did not pad its results was implemented in Co-dfns and used in the following implementations. In the following sections, we use ⌺ to mean the stencil *function* and not the stencil operator as it appears in Dyalog APL. See equation (6) for a stencil function implementation of equation (2) and equation (12) for the corresponding implementation of equation (3).

3.3 Neural Network Vocabulary

The original u-net paper uses five distinct operations to describe the network (see figure 1):

1. A 3×3 convolution with a ReLU activation function is used as the primary operation
2. A copy and crop operation is used to transfer data across one row of the network
3. Max pooling layers on a 2×2 window are used to compute "down" the network
4. A 2×2 transposed convolution goes back "up" the network
5. The final output has a single 1×1 convolution with a soft-max layer

In our implementation, we mirror this vocabulary by implementing the forward and back functions for each of these layers, one for each of the above operations. This results in a total of 10 functions grouped into 5 pairs, which we will take in turn.

3.3.1 Convolution (3×3) with ReLU.

The primary u-net convolutional layer is a 3×3 convolution with a ReLU activation function. The convolution in the paper uses "valid" convolutions, meaning that no padding is used. This implies that the convolution dimensions of the output array shrink by 2 for each dimension compared to the input. We define the forward propagation function CV as a function over a set of kernels **α** and a layer **ω** that obeys the following shape invariant:

$$\rho\,\alpha\,CV\,\omega \quad\leftrightarrow\quad (^{-}2+2\uparrow\rho\omega),{}^{-}1\uparrow\rho\alpha \tag{5}$$

We write $S\uparrow A$ to describe the array derived from A whose shape is equal to the shape of A except that the leading dimensions of $S\uparrow A$ are $|S$ (absolute value over S), read as "the S take of A." Negative values in S take from the "far" or "trailing" side of the dimension. Thus, the resulting shape of $\alpha\,CV\,\omega$ is the leading dimensions of the input ω subtracted by 2 catenated with the final dimension (the output channels) of kernel α. In the case of a u-net layer, we have input kernels of shape $3\,3\,I\,O$ and input layer of shape $N\,M\,I$ where $N\,M$ are the image/layer dimensions, and $I\,O$ are the input and output channel counts, respectively. The resulting output layer has shape $(N-2)\,(M-2)\,O$.

Using the stencil function, we define CV as follows for rank 4 kernel inputs and rank 3 layer inputs:

$$CV \quad\leftarrow\quad \{0\lceil(,\text{⍤}3\vdash 3\,3\text{⌺}\omega)+.\times,[\iota 3]\alpha\} \tag{6}$$

We include the ReLU function $0\lceil\omega$ as the final operation following the convolution. We write $,\text{⍤}3\vdash\omega$ to describe the value ω with its 3 trailing dimensions collapsed into a single dimension. We write $,[\iota 3]\omega$ to describe the value ω with its leading 3 dimensions likewise collapsed. In 2.3.3 we presented $+.\times$ as the matrix product form of the operator inner product, and in CV we use its extension to arrays of arbitrary rank.

Inside of the u-net architecture itself, we want to save the output of the convolution and the input to facilitate the backpropagation pass, and we obtain our kernels from a single source containing all network kernels. This results in the following source code implementation of CV that does the appropriate saving of layers and extracting of kernel data given a label index into the network as its left argument instead of a kernel, where Z is our storage for back propagation and W contains the weights:

```
CV←{
  z←(,⍤3⊢3 3⌺⊃Z[α]←⊂ω)+.×,[⍳3]α⊃W
  0⌈z⊣Z[α]←⊂Z[α],⊂z
}
```

Computing the backpropagation uses very similar approaches. Given the output layer z, input layer x, activation layer a, weights α, and the gradient backpropagated so far ω, we compute the transposed weights w, the derivative output layer Δz , the weight gradient Δw, padded output layer ΔZ, and the resulting back gradient Δx as follows:

$$\begin{aligned}
w &\leftarrow \ominus\,\text{⌽}\,[1]0\,1\,3\,2\,\text{⍉}\,\alpha \\
\Delta z &\leftarrow \omega\times 0<a && (7)\\
\Delta w &\leftarrow 3\,0\,1\,2\,\text{⍉}\,(\text{⍉},[\iota 2]\Delta z)+.\times,[\iota 2]3\,3\text{⌺}x && (8)\\
\Delta Z &\leftarrow {}^{-}2\ominus{}^{-}2\,\text{⌽}\,[1](4+2\uparrow\rho\Delta z)\uparrow\Delta z \\
\Delta x &\leftarrow (,\text{⍤}3\vdash 3\,3\text{⌺}\Delta Z)+.\times,[\iota 3]w && (9)
\end{aligned}$$

Since our stencil function does not pad its results, the expression $^{-}2\ominus{}^{-}2\,\text{⌽}\,[1](4+2\uparrow\rho\Delta z)\uparrow\Delta z$ expands the shape of Δz to ensure that the output convolution dimensions are 2 greater than those of Δz, where the functions $\ominus$ and ⌽ are functions to rotate an array. The function ⌽[1] rotates an array along the $1st$ dimension while $\ominus$ rotates along the leading axis. The resulting function ΔCV is written as follows:

```
∆CV←{
  x←⊃α⊃Z ⋄ a←1⊃α⊃Z ⋄ k←α⊃W
  w←,[⍳3]⊖⌽[1]0 1 3 2⍉k
  ∆z←ω×0<a
```

```
    ΔZ←¯2⊖¯2⌽[1](4+2↑⍴Δz)↑Δz
    Δw←⍺ Δ 3 0 1 2⍉(⍉,[⍳2]Δz)+.×,[⍳2]3 3⌺x
    Δx←w+.×⍨,[2+⍳3]3 3 SF ΔZ
}
```

In the above code, we have a function Δ that updates the weights in W, described in section 3.4.

3.3.2 Copy and Crop. Conceptually, the copy and crop operation is the simplest of the functions in u-net. Its sole job is to take the output from one side of the U-shaped net and move it over to the other side, adjusting the dimensions to ensure that it fits. In the forward direction, the input layer will have a greater dimension than the output layer, so we crop as evenly as possible around the edges and then catenate the result at the head of the layer coming "up" from the network to form the output layer with twice the channels of the "up" layer. The following function CC computes the crop of α catenated with ω using $\downarrow$ (read as "drop"), which is the opposite function of the previously described $\uparrow$ ("take") function:

$$CC \quad \leftarrow \quad \left\{ \begin{matrix} p \leftarrow ((\rho\alpha) - \rho\omega) \div 2), \omega \\ ((\lfloor p) \downarrow (-\lceil p) \downarrow \alpha \end{matrix} \right\} \tag{10}$$

For dimensions that are not evenly divisible by two, we choose to round up on the right and bottom sides and round down on the left and upper sides of the layer. Computing the backpropagation of CC given the input α and output gradient ω simply reverses this operation and expands the shape back to the original input size. This result is then added to the appropriate layer in the u-net architecture described in section 3.4.

$$\begin{aligned} n\,m & \leftarrow -\lfloor(2 \uparrow (\rho\alpha) - \rho\omega) \div 2 \\ \Delta x & \leftarrow n \ominus m \varphi [1](\rho\alpha) \uparrow \omega \end{aligned} \tag{11}$$

This leads to the following code for the forward and backpropagation passes:

```
CC←{
    ⍵,⍨(⌊p)↓(-⌈p)↓(⍺⊃Z)⊣p←2÷⍨(⍴⍺⊃Z)-⍴⍵
}

ΔCC←{
    x←⍺⊃Z ⋄ Δz←⍵ ⋄ d←-⌊2÷⍨2↑(⍴x)-⍴Δz
    (⊃d)⊖(1⊃d)⌽[1](⍴x)↑Δz
}
```

3.3.3 Max Pooling. Max pooling is a shrinking convolution that computes the maximum value in a non-overlapping sliding window. Given the stencil function, the max pool over a layer is given by the following expression:

$$\lceil\!\not/[2], [2\,3](2\,2\rho 2)⌺\omega \tag{12}$$

Here we write $\lceil\!\not/[2], [2\,3]\omega$ to describe an array where we have collapsed dimensions 2 and 3 and computed the maximum value reduction over the resulting dimension. For example, given an input layer ω of shape $N\,M\,C$, the result of $(2\,2\rho 2)⌺\omega$ is a rank 5 array of shape $(N \div 2)(M \div 2)2\,2\,C$. We then collapse the $2nd$ and $3rd$ dimensions to form an array of shape $(N \div 2)(M \div 2)4\,C$ and subsequently find the maximum value for each vector along the $2nd$ dimension, resulting in an array of shape $(N \div 2)(M \div 2)C$.

Computing the backpropagation of this involves replicating each of the stencil dimensions, which are the two leading axes in our implementation. We write $n\!\not/A$ and $n/[1]A$ to indicate the array A with each element duplicated or repeated along the first and second axes, respectively, n times. Given an input α and output layer ω the following expression computes the backpropagation:

$$y \times \alpha = y \leftarrow (\rho\alpha) \uparrow 2\!\not/2/[1]\omega \tag{13}$$

This leads to the following linked source implementation for max pooling:

```
MX←{
    ⌈⌿[2],[2 3](2 2⍴2)⌺⊃Z[⍺]←⊂⍵
}

ΔMX←{
    x←⍺⊃Z ⋄ Δz←⍵
    y×x=y←(⍴x)↑2⌿2/[1]Δz
}
```

3.3.4 Transposed Convolution (2×2). In the initial exploration of this implementation, the upsampling computation with convolution proved to be the most subtle and challenging, mostly in part to the opaqueness of implementations. The u-net paper was not immediately transparent regarding the exact operations used for this layer and there were a number of potential design decisions that could have been made. Moreover, for users reading about upsampling through convolutions, the descriptions are also the furthest removed from a reasonable implementation of the same. However, once the intuition of how an upsampling convolution matches the shape and form of a non-overlapping sliding window in the output layer, expressed via the simple expression $K \subset ⍤ \times ⍤ 2\,0 \vdash A$, the computation becomes much clearer.[13]

For this convolution, we change the anticipated kernel shape from that used for CV above. Whereas CV expects kernels of shape $3\,3\,I\,O$, our transposed convolutions expect kernels of shape $I\,2\,2\,O$ for input channels I and output channels O. Given a layer of our standard shape $N\,M\,I$, this gives the following definition for the upsampling pass.:

$$UP \quad \leftarrow \quad \{,[\iota 2]\,,[2\,3]\rho 0\,2\,1\,3\,4 \oslash \omega +.\times \alpha\} \tag{14}$$

[13] https://mathspp.com/blog/til/033

The key change here from the reliance on $+.\times$ with CV is the use of a dyadic transpose $0\ 2\ 1\ 3\ 4 \oslash$. Dyadic transpose is sometimes considered a somewhat challenging concept in APL. In brief, for a rank r array A of shape S and targeting array D where $(\rho D) \equiv \rho S$, we write $D \oslash A$ to describe a transposed array with shape T where $T[D] \leftarrow S$, assuming that $\wedge\!\!\not/ D \in \iota\rho S$, that is, all dimensions of S are mentioned in D. So, given a targeting array $0\ 2\ 1\ 3\ 4$ and an input array A of shape $N\ M\ 2\ 2\ O$, the expression $0\ 2\ 1\ 3\ 4 \oslash A$ describes an array with elements from A of shape $N\ 2\ M\ 2\ O$. As the final operation, we collapse the first two pairs of leading dimensions, giving a final output array of shape $(N{\times}2)(M{\times}2)O$.

To compute the backpropagation pass, we compute the convolutions on a 2×2 sliding window with stride 2.

$$\Delta w \quad \leftarrow \quad (\oslash,[\iota 2]x) +.\times ,[\iota 2](2\ 2\rho 2)\text{⌺}\Delta z \tag{15}$$

$$\Delta x \quad \leftarrow \quad (,[2+\iota 3](2\ 2\rho\ 2)\text{⌺}\Delta z) +.\times \oslash \text{⍪}\alpha \tag{16}$$

This gives the following source implementations for transposed convolutions:

```
UP←{
  Z[α]←⊂ω
  ,[⍳2],[2 3]⍴0 2 1 3 4⍉ω+.×α⊃W
}

ΔUP←{
  w←α⊃W ⋄ x←α⊃Z ⋄ Δz←ω ⋄ cz←(2 2⍴2)⌺Δz
  Δw←α Δ(⍉,[⍳2]x)+.×,[⍳2]cz
  Δx←(,[2+⍳3]cz)+.×⍉⍪w
}
```

3.3.5 Final 1×1 Convolution.

The final convolution is a 1×1 convolution with 2 output channels, which means that it collapses the final incoming channels into an output layer with only two channels. This gives the trivial simplification of our convolution code over layer ω and kernel α:

$$\omega + . \times \alpha \tag{17}$$

Additionally, the paper describes using a soft-max layer, which we include at this phase:

$$1e^{-}8 + z \div [\iota 2]+/z \leftarrow *\omega - [\iota 2]\lceil/\omega \tag{18}$$

Computing the backpropagation is likewise a simplification of the more complex CV code:

$$\Delta w \quad \leftarrow \quad (\oslash,[\iota 2]x)+.\times,[\iota 2]\Delta z \tag{19}$$

$$\Delta x \quad \leftarrow \quad \Delta z +.\times \oslash w \tag{20}$$

Which leads to the following source implementations:

```
C1←{
  Z[α]←⊂ω
  1E¯8+z÷[⍳2]+/z←*z-[⍳2]⌈/z←ω+.×α⊃W
}

ΔC1←{
   w←α⊃W ⋄ x←α⊃Z ⋄ Δz←ω
  Δw←α Δ(⍉,[⍳2]x)+.×,[⍳2]Δz
  Δx←Δz+.×⍉w
}
```

3.4 U-net Architecture

Given the core vocabularies defined in section 3.3, the remaining challenge with implementing u-net is to link together the appropriate layers and compositions to form the complete network as described by figure 1. To do this, we observe that the structure of the u-net diagram is an almost symmetric pattern. The output layer computations form 3 operations which are not part of the pattern, but the rest of the pattern decomposes into 4 depths, each with 6 operations each. table 2 contains a visual arrangement of the kernel shapes used in our architecture mirroring the overall structure of figure 1.

Additionally, we note that the U-shaped structure also mimicks the down and up nature of a recursive program call-tree. Thus, our overall strategy is to implement a recursive function LA that receives an index identifying a particular depth of the network, computes the appropriate "downward pass" operations before recuring deeper into the network and finally computing the upwards passes on the return of its recursive call. We likewise implement backpropagation in same way, but in the opposite direction. Assuming that α contains the computed depth offset for the network layer, we write $\alpha + i$ to access the ith column of the network described in table 2 at the depth $\alpha \div 6$.

Our forward pass function is responsible for initializing an appropriate holding place for the intermediate results produced by forward propagation for use by the backpropagation function. Additionally, after the recursive computation, there are the final three operations, $C1$ and two CV operations, that must be called before returning. We also assume that we may receive a rank 2 matrix instead of a rank 3 layer as input, and so we reshape the input to ensure that we always have a rank 3 input to LA. This gives us the following function definition:

```
FWD←{
  Z⊢←(≢W)⍴⊂⍬
  ⍝ Forward propagation layers ...
  LA←{
    α≥≢Z:ω
    down←(α+6)∇(α+2)MX(α+1)CV(α+0)CV ω
    (α+2)CC(α+5)UP(α+4)CV(α+3)CV down
  }
  2 C1 1 CV 0 CV 3 LA ω⍴⍨3↑1,⍨⍴ω
}
```

	Operation					
	Downward Pass			Upward Pass		
Depth	*CV*	*CV*	*MX*	*CV*	*CV*	*UP*
0	3 3 1 64	3 3 64 64	0 0 64 64	3 3 256 128	3 3 128 128	128 2 2 64
1	3 3 64 128	3 3 128 128	0 0 128 128	3 3 512 256	3 3 256 256	256 2 2 128
2	3 3 128 256	3 3 256 256	0 0 256 256	3 3 1024 512	3 3 512 512	512 2 2 256
3	3 3 256 512	3 3 512 512	0 0 512 512	3 3 512 1024	3 3 1024 1024	1024 2 2 512

Table 2. A rectangular arrangement of the u-net network

The backwards computation mirrors this pattern, except that it proceeds in the opposite direction and also defines an updater function Δ that will update the network weights in W and the velocities in V based on a given gradient ω and index α pointing to a specific location in the network.

```
BCK←{
  Y←α ⋄ YΔ←ω
  Δ←{
    V[α]←⊂ω+MO×(⍴ω)⍴α⊃V
    W[α]←⊂(α⊃W)-LR×α⊃V
  }
  ⍝ Back propagation layers ...
  ΔLA←{
    α≥≢Z:ω
    d←{(α+3)ΔCV(α+4)ΔCV(α+5)ΔUP ω}
    c←{(α+2)ΔCC ω}
    u←{(α+0)ΔCV(α+1)ΔCV(α⊂ω)+(α+2)ΔMX}
    α u (α+6)∇ α d ω↑[2]⍨-2÷⍨⊃⌽⍴ω
  }
  3 ΔLA 0 ΔCV 1 ΔCV 2 ΔC1 YΔ-(~Y),[1.5]Y}
```

We also need to compute an error over the soft-max computed by *FWD*. This is given by the following function, which is based off of the error function given in the original u-net paper by Ronneberger et al. [2015].

```
E←{-+⌿,⍟(α×ω[;;1])+(~α)×ω[;;0]}
```

Finally, we wire all of these functions together into a *RUN* function that runs the forward pass and backward pass functions and returns three values, the expected inputs Y, the computed results $Y\Delta$ from *FWD*, and the error given by $Y\,E\,Y\Delta$. We reshape the original reference input to match the size of $Y\Delta$.

```
RUN←{
  YΔ←FWD α
  Y←⌊0.5+nm↑ω↓⍨2÷⍨(⍴ω)-nm←2↑⍴YΔ
  Y YΔ(Y E YΔ)⊣Y BCK YΔ
}
```

4 Performance

To examine the performance profile of our APL implementation, we primarily focused on comparing our u-net implementation against a reference implemented in PyTorch [Paszke et al. 2019], which is an easy to use Python framework with good performance. In addition to this primary performance analysis, we examined the performance of two varieties of stencil computations within the APL language. We also make note of some small exploratory effects that we discovered while implementing the stencil function and convolutions in Co-dfns.

4.1 U-net Performance

We were primarily interested in the costs of executing a single run of the u-net over a single image source in both the forwards and backwards directions. We compared performance over the following platforms:

- Dyalog APL 18.0 64-bit Windows interpreter
- Co-dfns (v4.1.0 master branch) using the CUDA backend (AF v3.8.1)
- Co-dfns (v4.1.0 master branch) using the CPU backend (AF v3.8.1)
- PyTorch v1.10.2 with the CUDA gpu backend
- PyTorch v1.10.2 with the multi-threaded CPU backend
- PyTorch v1.10.2 with the single-threaded CPU backend

The results of the execution can be seen in figure 2. The timings do not include the cost of reading the image data from disk, but they do include the costs of transferring the image input data and the resulting error and forward propagation results back to the CPU main memory. In our testing, data transfer costs in Co-dfns accounted for significantly less than 5% of the total runtime.

The hardware used was an NVIDA GeForce RTX 3070 Laptop GPU with 8GB of dedicated graphics memory. We used NVIDIA driver version 511.65. The CPU was an Intel Core i7-10870H with 16 logical cores @ 2.2GHz. Main system memory was 32GB of DDR4 RAM. The system was running an up to date release of Microsoft Windows 11.

As input we used the original image data from the ISBI benchmark referenced in the u-net paper [Cardona et al. 2010; Ronneberger et al. 2015]. These images are 512×512 images in grayscale with a binary mask for training. Each run took one of these images and associated training mask

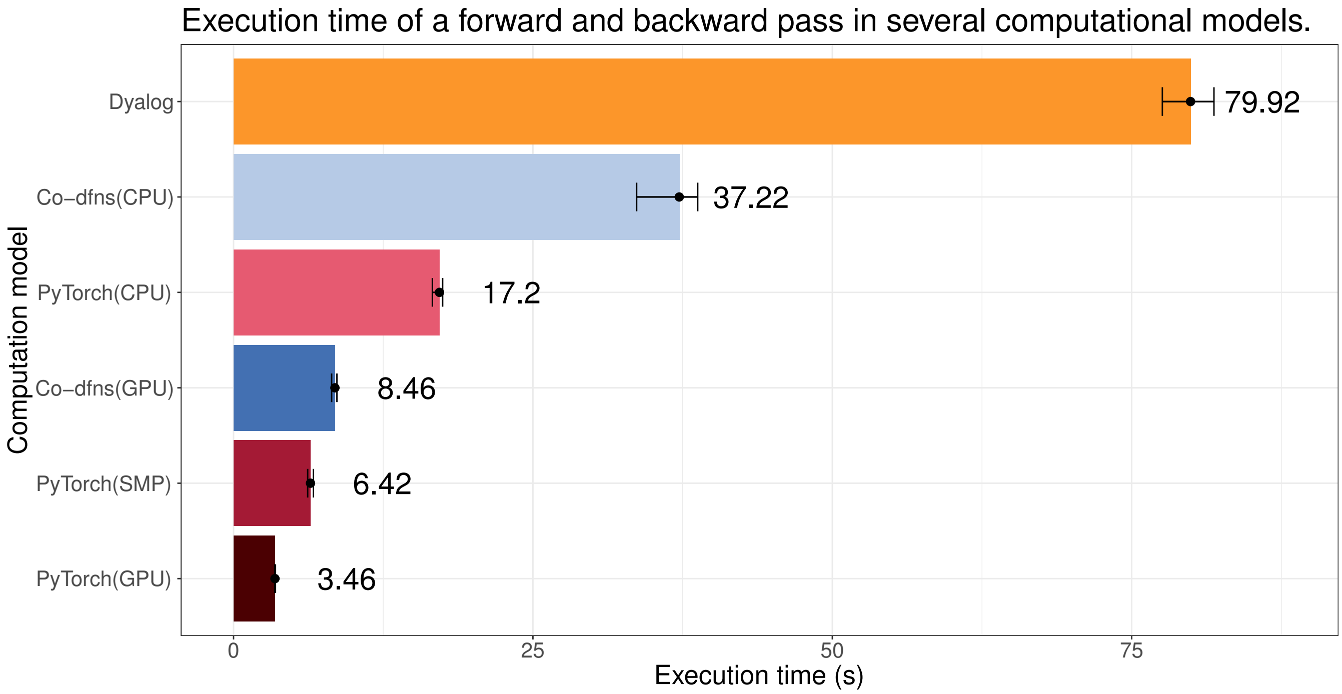

Figure 2. Performance results for U-net across a range of platforms

	0	1	2	3	4	5	6	7	8	9	*Avg*
Dyalog	80.90	77.56	78.32	81.33	77.95	80.51	81.08	81.87	80.23	79.42	79.92
Co-dfns (CPU)	34.11	33.66	35.98	37.72	38.20	38.64	38.11	38.46	38.57	38.76	37.22
Co-dfns (GPU)	8.39	8.20	8.62	8.63	8.49	8.39	8.19	8.63	8.51	8.59	8.46
PyTorch (CPU)	17.27	17.34	17.21	16.60	17.46	17.01	17.33	17.04	17.36	17.33	17.22
PyTorch (SMP)	6.31	6.42	6.45	6.19	6.67	6.49	6.43	6.38	6.36	6.55	6.42
PyTorch (GPU)	3.50	3.50	3.47	3.44	3.44	3.44	3.44	3.44	3.46	3.46	3.46

Table 3. Raw timings for each computing platform in seconds

and computed the result of forward and backwards propagation and the error as well as updating the weights for the network.

When working on the network, APL implementations generally do not have a concept of small floating point values. Rather, their default is to always use 64-bit floating point values when floats are called for. In order to try to mimic this behavior as closely as possible, we attempted to feed 64-bit data into the PyTorch models. However, because of the opaqueness of the PyTorch implementation, we were not able to fully verify that 64-bit values are used throughout the PyTorch computational network. On the other hand, the reliance on 64-bit only floating points, while a boon to convenience and user-friendliness for non-computer science programmers, creates well-defined performance issues for an application like this.

When running the benchmark, we computed the average of 10 runs, ensuring that we discarded the first run each time, since these runs often contained significant setup and bootstrapping code (PyTorch's optimizer, the JIT optimization in Co-dfns, and so forth). The figure includes information about the variance of the individual runs as well as the average run time in seconds.

Examining the data, it is clear why traditional APL implementations were relatively unsuited to extensive use within the machine learning space. Dyalog's interpreter preformed the slowest by a very large magnitude. After this, the single-threaded CPU implementations in Co-dfns and PyTorch are predictably the next slowest, with the Co-dfns implementation running about a factor of 2.2 times slower than the equivalent PyTorch implementation.

When acceleration techniques are employed, the differences in execution speed begin to shrink, with PyTorch's multi-threaded and GPU-based implementations coming in fasted, and Co-dfns' GPU backend running at roughly 2.4 times slower than the PyTorch GPU execution.

We observed the widest variance in performance results in the Co-dfns CPU and Dyalog interpreter-based runs, and very little variance in the GPU-based runs or in PyTorch itself.

4.1.1 Co-dfns Runtime Implementation.

The stencil function was modeled in APL and used to conduct the above benchmark. The model, written in APL, is a naive implementation of the stencil function and contains no special optimizations other than to distinguish between sliding windows of step 1 and non-overlapping, adjacent windows (such as used for the max pooling layers). Additionally, no specialized code was used within Co-dfns that was specific or specialized to neural network programming.

The above benchmark therefore represents a comparison of the PyTorch implementation against a naive and unspecialized implementation in APL executed with the general-purpose runtime used in Co-dfns that provides generalized GPU computation but does not include domain-specific optimizations such as those available in PyTorch.

4.2 APL Stencil Primitives

In this section we present some benchmarks relative to the speedup we get when considering convolutional layers and max pooling layers that are based on the stencil function instead of the stencil operator.

Figure 3 compares convolutional layers based on the stencil function with convolutional layers based on the stencil operator. To produce this benchmark, we consider inputs of different sizes and number of channels. Then, for each value *size* and number *channels*, we create a random input cuboid of shape *size size channels* that is passed through a convolutional layer with a kernel having shape *channels* 3 3 *channels*, which means the output also has shape *size size channels*. After creating these inputs and kernels, we benchmark the runtime of the two convolutional layers (one stencil operator-based and one stencil function-based) and divide the runtime of the convolutional layer based on the stencil function by the runtime of the convolutional layer based on the stencil operator, in order to compute the speedup we get by adopting the stencil function.

Similarly, Figure 4 compares max pooling layers based on the stencil function with max pooling layers based on the stencil operator. The experimental setup is identical, except that we do not have to generate random kernels for the max pooling layers.

While Figure 3 shows a significant speedup achieved through the introduction of the stencil function in the convolutional layers, Figure 4 shows that the max pooling layer based on the stencil function is only slightly faster when the number of channels is small, becoming slightly slower than the convolutional layer based on the stencil operator when the number of channels increases.

4.3 Microbenchmarks Against Other Libraries

It is worth noting that we also explored optimizing our u-net with specialized functions available via the Co-dfns platform (specific max filters and convolutions) that are domain-specific operations much like those available in PyTorch. This sort of operation can be done via "idiom recognition" and for our small convolution expressions, it is quite conceivable that idiom recognition could apply and convert these functions to use domain-specific code under the hood for convolutions and maximum filters, &c.

However, we aborted this line of inquiry for now because in microbenchmarking the domain-specific functions against our naive implementation of the stencil function, we discovered that the domain specific functions were actually a factor of 2 *slower* than our naive implementations in our primary sample inputs. Given that the underlying specialized functions are supposed to wrap the CuDNN library [Yurkevitch 2020], it is surprising that we achieved faster results with the naive stencil function implementation over the domain specific implementations available within the general purpose array libraries leveraged by Co-dfns.

We are not sure of the cause of these slowdowns, and therefore, we did not include a formal benchmark of these results here. It is possible that a misconfiguration or some other element is causing these degradations, and so we intend to explore these performance issues in more detail, but we wished to at least note this effect, since this represents a different result from our benchmarking here, which tests Co-dfns against a specialized framework, rather than a more generalized array framework with specialized functions within it. The specialized frameworks appear to be clearly faster at the moment than the naive Co-dfns implementations, but this is not true thus far in our limited testing of specialized functions exposed within more generalized array libraries.

5 Discussion

5.1 Pedagogy

Pedagogy is a concern for new approaches to solving problems both in academic as well as industrial spaces. We believe that APL is a double-edged sword in this regard. On one hand, there is significant institutional momentum around languages like Python. This creates a large base of prior knowledge which can be leveraged by new users. This results in users feeling like learning a Python based framework is easier than learning a whole new language, and they are probably right, in the short term.

However, it has been argued [Iverson 2007] that APL has two distinct advantages from a pedagogical point of view that may warrant more interest. First, what one learns in APL tends to also have direct skills transferrence to many other programming domains, whereas in a more domain-specific, library-centric approach, learning the particular API for one domain often does not transfer any skills beyond

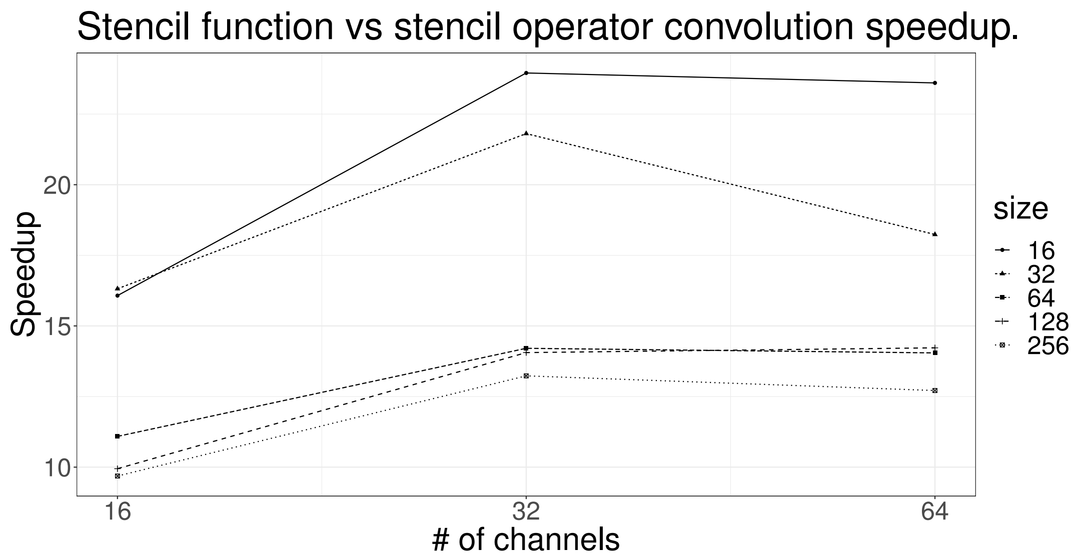

Figure 3. Speedup of stencil function-based convolution with respect to stencil operator-based on inputs of shape *size* × *size* × *channels* and kernels of size *channels* × 3 × 3 × *channels*.

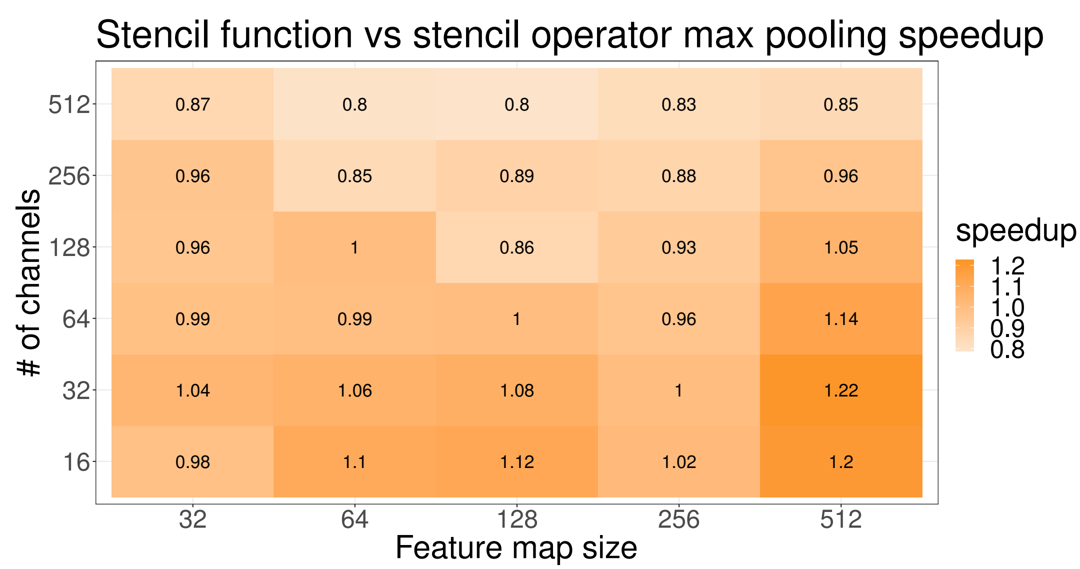

Figure 4. Speedup of stencil function-based max pooling with respect to stencil operator-based on inputs of shape *size* × *size* × *channels*.

that domain directly. In the case of u-net, all of the operations used to build the u-net system are general array programming concepts that are widely applicable to many other domains, and are not restricted solely to convolutional neural networks.

Second, a transparent and direction implementation of u-net in APL is somewhat uniquely compact and simple, making it much easier to not only delve deeper into the CNN domain, but also to make adjustments and modifications to taste as one becomes more experienced. Starting with a transparent implementation enables programs to be enhanced or adapted or optimized without requiring the inclusion of abstractions that increase program indirection and opaqueness. The notational aspects of APL facilitates this sort of expressive power in a way that other languages do not, especially from a "human factors" perspective.

However, the current learning materials for APL, particularly in a space like neural networks, are clearly underdeveloped and in need of improvement. We believe it is likely possible to make it as easy to reference a convolutional implementation in APL as an API reference for PyTorch, but the current ecosystem is not there yet.

5.2 Performance

Clearly, specialized frameworks for deep neural networks are still the best way to go in order to achieve the absolute maximum in performance at present. However, our results indicate that the gap between reliance on specialized frameworks and the freedom to use more general purpose and transferrable programming languages while still achieving competitive performance is not nearly as large as might have been the case even a few years ago.

Given that almost zero special optimization is taking place for the APL implementation executed under the Co-dfns runtime, it is impressive that we are able to see results that come close to a factor of 2 of the specialized frameworks. Given some of the obvious design issues that would contribute to slower performance, it seems more reasonable to be able to expect more general purpose languages like APL to be able to reach performance parity with specialized frameworks, without the requirement that the user learn a special API, or import specialized dependencies. In more complex applications that leverage APL for other domain-intensive work, this suggests that APL might facilitate scaling such applications to integrate machine learning algorithms more easily and with less programmer effort than might be required to integrate a separate framework like PyTorch.

5.3 Stencil Operator

The results we found regarding stencil operator performance vs. stencil function performance suggest to us that companies like Dyalog or implementors of Co-dfns should focus on implementing and improving the performance of a stencil function instead of continuing with the use of a stencil operator, which suffers from a number of design issues that makes it incongruous with the rest of an otherwise elegantly designed language.

It is likely that scalable performance with the stencil function will be easier to achieve and easier to maintain over the long term. Moreover, the stencil function results in more compositional code that it easier to work with using the rest of the APL language than the stencil operator.

5.4 APL vs. Frameworks

We have demonstrated that APL itself, without libraries or additional dependencies, is exceptionally well suited to expressing neural network computations, at a level of inherent complexity that is arguably equal or possibly even less than that of the reference PyTorch implementation. At the very least, it is less code with less underlying background code and layers. This comes at the admittedly heavy cost of being completely foreign and alien to most programmers who are more familiar with languages like Python. This certainly creates a fundamental and immediate learning cost to APL over other frameworks, since other frameworks can assume a wider range of pre-knowledge around their chosen language implementation.

It remains unclear, however, whether, if this pre-knowledge were taken away, APL would represent a compelling value proposition for such programming tasks or not. Indeed, it is exceptionally challenging to divorce the present reality of prior background knowledge from such a question. Even fundamental knowledge like what it means to do array programming and how to structure problems in an array-style are rarely if ever taught at universities, whereas most classes spend significant amounts of time teaching students how to utilize the Python-style programming model of PyTorch.

The argument that APL may be used more widely and broadly than PyTorch on a wider range of problems using the same skillset may not matter to users who are only interested in deep learning algorithms.

APL presently has a higher barrier to entry, but rewards the user with full and effortless control over what's being done in a way that other systems do not. This may present itself as a distinct advantage to users who are looking to expand "off the beaten track" and utilize novel approaches that do not easily fit within existing frameworks.

We encountered significant difficulties in identifying exactly what the original authors did based on their paper alone because of many implementation details that were omitted. On the other hand, APL enables us to express our entire implementation in a way that makes every implementation detail clear, improving the ability of others to reproduce our work.

Finally, in the implementation of u-net in APL, we gained insights into the architecture that had a direct and material influence on the PyTorch reference implementation that would not have emerged without first having studied the APL implementation. Thus, we gained significant insight simply from doing the APL implementation, even if we were to re-implement that code in PyTorch.

6 Related Work

Two particular avenues of research warrant particular mention here. In addition to the Co-dfns compiler, Šinkarovs et al. [2019] have explored alternative implementations to CNNs, though not u-net specifically. They focused specifically on APL as a productivity enhancement for CNN development, and only benchmarked the APL implementation on the CPU using the Dyalog APL interpreter. However, they indicated work in progress on a compiled version using the APEX compiler with a SaC backend. Their conclusion regarding the performance of APL-based systems may have been premature given the results we found here, but they make a case for the usability of APL even with the performance numbers they achieved. While their code exhibits some material differences to that given here, there are nonetheless some similarities that demonstrate some level of convergence around implementing CNNs in APL.

Another approach to GPU-based array programming with an APL focus is the TAIL/Futhark system [Henriksen et al. 2017], which is a compiler chain taking APL to the TAIL (Typed Array Intermediate Language) and then compiling TAIL code using the Futhark GPU compiler backend. While the authors are not aware of any work implementing complex neural networks with this chain, it represents an interesting approach to compilation of APL via typed intermediate languages, which have the potential to enhance the fusion that can be done with an operation like the stencil function.

Other programming environments that are often categorized as array programming environments, such as Matlab [Math Works 1992], Julia [Bezanson et al. 2017], and Python/Numpy [Harris et al. 2020; Van Rossum and Drake 2009], are not exceptionally performant on their own for machine learning, but often wrap libraries to do so. Unfortunately, many of these languages use a syntax that much more closely mirrors that of Python than APL. In our perspective, this reduces the value proposition of such languages over using specialized frameworks, since one does not obtain the particular clarity and productivity benefits associated with the APL notation.

7 Future Work

One of the most obvious questions to answer in future work is the reason for the slower performance of the specialized convolution functions against our naive implementation when using the same backend in Co-dfns.

There are a number of design elements of the current crop of APL implementations, including Co-dfns, which hamper performance for machine learning. Especially, the use of 64-bit floating points without any feature to reduce their size makes memory usage a concern. Additionally, no optimization on the design of stencil has been done, while optimizations related to lazy indexing, batch processing, and a number of other features seem readily accessible.

Additionally, we would like to explore the potential of using such systems to improve machine learning pedagogy by encouraging students to have access to high-performance, but also transparent, implementations of foundational machine learning concepts. There are still some challenges to recommending this approach at scale for a large number of educational institutions, but we believe work on understanding the pedagogical benefits of APL warrants further research in addition to exploring APL in the professional space.

8 Conclusion

Given the notational advantages of APL and the concision and clarity of expression that one can obtain, we explored the potential impact of using APL as a language for implementing convolutional neural networks of reasonable complexity. We found that, though the traditional implementations of APL suffer from performance issues that would prevent widespread use in either academic, educational, or industrial contexts, compilers such as Co-dfns are capable of compiling complete neural network programs (in our case, the u-net architecture) and producing much more competitive performance results (within a factor of 2.2 - 2.4 times of our reference PyTorch implementation). This is despite the naive nature of our implementation and the naive optimization support for neural networks on the part of the Co-dfns compiler.

Furthermore, we found that our effort to implement u-net in APL resulted in a concise but fully unambiguous implementation that provided transparency over the entire source, without any frameworks or library dependencies. Despite being a complete "by hand" implementation, its complexity of expression is on par with that of PyTorch and other specialized frameworks, or even better, particularly in cases where more exploration and novel implementation is required, or when customized integrations may be called for. The insights that we gained from implementing u-net in APL affected our implementation of a reference implementation in PyTorch directly, suggesting that APL may have significant pedagogical advantages for teaching neural network programming and machine learning in general.

References

Manuel Alfonseca. 1990. Neural Networks in APL. *SIGAPL APL Quote Quad* 20, 4 (may 1990), 2–6. https://doi.org/10.1145/97811.97816

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

Albert Cardona, Stephan Saalfeld, Stephan Preibisch, Benjamin Schmid, Anchi Cheng, Jim Pulokas, Pavel Tomancak, and Volker Hartenstein. 2010. An Integrated Micro- and Macroarchitectural Analysis of the Drosophila Brain by Computer-Assisted Serial Section Electron Microscopy. *PLOS Biology* 8, 10 (10 2010), 1–17. https://doi.org/10.1371/journal.pbio.1000502

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.

Pedro Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87.

Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571.

Aaron Wen-yao Hsu. 2019. *A data parallel compiler hosted on the gpu*. Ph. D. Dissertation. Indiana University.

Roger Hui. 2017. Stencil Lives. https://www.dyalog.com/blog/2017/07/stencil-lives/

Roger Hui. 2020. Towards Improvements to Stencil. https://www.dyalog.com/blog/2020/06/towards-improvements-to-stencil/

Roger KW Hui and Morten J Kromberg. 2020. APL since 1978. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 1–108.

Kenneth E Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. 345–351.

Kenneth E Iverson. 2007. Notation as a tool of thought. In *ACM Turing award lectures*. 1979.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

JSoftware. 2014. Vocabulary/semidot. https://code.jsoftware.com/wiki/Vocabulary/semidot

D Knuth. 1993. Computer literacy bookshops interview. *Also available as http://yurichev. com/mirrors/C/knuth-interview1993. txt* (1993).

Donald E Knuth. 2007. Computer programming as an art. In *ACM Turing award lectures*. 1974.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

Bernard Legrand. 2009. *Mastering Dyalog APL* (1 ed.). Dyalog Ltd.

Inc Math Works. 1992. *MATLAB reference guide*. Math Works, Incorporated.

Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378* (2018).

Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

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

Dominik Scherer, Andreas Müller, and Sven Behnke. 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*. Springer, 92–101.

Rodrigo Girão Serrão. 2022. Transposed convolution. https://mathspp.com/blog/til/033#transposed-convolution

Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional neural networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 69–79.

Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

Stefan Yurkevitch. 2020. ArrayFire v3.7.x Release. https://arrayfire.com/blog/arrayfire-v3-6-release-2/

Appendix A: Complete APL U-net implementation

```
:Namespace UNET

 W←θ ⋄ V←θ ⋄ Z←θ ⋄ LR←1e¯9 ⋄ MO←0.99

 FWD←{Z⊢←(≢W)⍴⊂θ
  CV←{0⌈z⊣Z[α]←⊂Z[α],⊂z←(,[2+⍳3]3 3⌺⊃Z[α]←⊂ω)+.×,[⍳3]α⊃W}
  CC←{ω,⍨(⌊p)↓(-⌈p)↓(α⊃Z)⊣p←2÷⍨(⍴α⊃Z)-⍴ω}
  MX←{⌈⌿[2],[2 3](2 2⍴2)⌺⊃Z[α]←⊂ω}
  UP←{((2×¯1↓⍴ω),¯1↑⍴α⊃W)⍴0 2 1 3 4⍉ω+.×α⊃W⊣Z[α]←⊂ω}
  C1←{1E¯8+z÷[⍳2]+/z←*z-[⍳2]⌈/z←ω+.×α⊃W⊣Z[α]←⊂ω}
  LA←{α≥≢Z:ω
    down←(α+6)∇(α+2)MX(α+1)CV(α+0)CV ω
    (α+2)CC(α+5)UP(α+4)CV(α+3)CV down}
  2 C1 1 CV 0 CV 3 LA ω⍴⍨3↑1,⍨⍴ω}

 BCK←{Y←α ⋄ YΔ←ω
  Δ←{0⊣W[α]←⊂(α⊃W)-LR×⊃V[α]←⊂ω+MO×(⍴ω)⍴α⊃V}
  ΔCV←{w←,[⍳3]⊖⌽[1]0 1 3 2⍉α⊃W ⋄ x←⊃α⊃Z ⋄ Δz←ω×0<1⊃α⊃Z
   ΔZ←¯2⊖¯2⌽[1](4+2↑⍴Δz)↑Δz
   _←α Δ 3 0 1 2⍉(⍉,[⍳2]Δz)+.×,[⍳2]3 3⌺x
   w+.×⍨,[2+⍳3]3 3⌺ΔZ}
  ΔCC←{x←α⊃Z ⋄ Δz←ω ⋄ d←-⌊2÷⍨2↑(⍴x)-⍴Δz ⋄ (⊃d)⊖(1⊃d)⌽[1](⍴x)↑Δz}
  ΔMX←{x←α⊃Z ⋄ Δz←ω ⋄ y×x=y←(⍴x)↑2⌿2/[1]Δz}
  ΔUP←{w←α⊃W ⋄ x←α⊃Z ⋄ Δz←ω ⋄ cz←(2 2⍴2)⌺Δz
   _←α Δ(⍉,[⍳2]x)+.×,[⍳2]cz
   (,[2+⍳3]cz)+.×⍉⍪w}
  ΔC1←{w←α⊃W ⋄ x←α⊃Z ⋄ Δz←ω ⋄ _←α Δ(⍉,[⍳2]x)+.×,[⍳2]Δz ⋄ Δz+.×⍉w}
  ΔLA←{α≥≢Z:ω
   down←(α+6)∇(α+3)ΔCV(α+4)ΔCV(α+5)ΔUP ω↑[2]⍨-2÷⍨⊃⌽⍴ω
   (α+0)ΔCV(α+1)ΔCV(ω ΔCC⍨α+2)+(α+2)ΔMX down}
  3 ΔLA 0 ΔCV 1 ΔCV 2 ΔC1 YΔ-(~Y),[1.5]Y}

 E←{-+⌿,⍟(α×ω[;;1])+(~α)×ω[;;0]}

 RUN←{Y YΔ(Y E YΔ)⊣(Y←⌊0.5+nm↑ω↓⍨2÷⍨(⍴ω)-nm←2↑⍴YΔ)BCK⊢YΔ←FWD α}

:EndNamespace
```

Appendix B: PyTorch Reference Implementation

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms.functional

class TwoConv(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.path = nn.Sequential(
            nn.Conv2d(in_channels, out_channels,
                kernel_size=(3, 3), bias=False),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels,
                kernel_size=(3, 3), bias=False),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.path(x)

class Down(nn.Module):

    def __init__(self, in_channels):
        super().__init__()

        self.path = nn.Sequential(
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            TwoConv(in_channels, 2 * in_channels),
        )

    def forward(self, x):
        return self.path(x)

class Up(nn.Module):

    def __init__(self, in_channels):
        super().__init__()

        self.upsampling = nn.ConvTranspose2d(
            in_channels,
            in_channels // 2,
            kernel_size=(2, 2),
            stride=2,
            bias=False,
        )
        self.convolutions =
            TwoConv(in_channels, in_channels // 2)

    def forward(self, x_to_crop, x_in):
```

```
        upped = self.upsampling(x_in)
        cropped = torchvision.transforms.functional.center_crop(
            x_to_crop, upped.shape[-2:]
        )
        x = torch.cat([cropped, upped], dim=1)
        return self.convolutions(x)

class USegment(nn.Module):

    def __init__(self, in_channels, bottom_u=None):
        super().__init__()

        # Default value for the bottom U.
        if bottom_u is None:
            bottom_u = lambda x: x

        self.down = Down(in_channels)
        self.bottom_u = bottom_u
        self.up = Up(2 * in_channels)

    def forward(self, x):
        return self.up(x, self.bottom_u(self.down(x)))

class UNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.u = USegment(512)
        self.u = USegment(256, self.u)
        self.u = USegment(128, self.u)
        self.u = USegment(64, self.u)
        self.path = nn.Sequential(
            TwoConv(1, 64),
            self.u,
            nn.Conv2d(64, 2, kernel_size=1, bias=False),
        )

    def forward(self, x):
        return self.path(x)
```