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 naïve 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.

CCS Concepts: • **Software and its engineering** → Parallel programming languages; Runtime environments.

Keywords: APL, Compilers, Neural Networks, Machine Learning, GPU, Co-dfns

ARRAY '23, June 18, 2023, Orlando, FL, USA
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0169-6/23/06...$15.00
https://doi.org/10.1145/3589246.3595371

ACM Reference Format:
Aaron W. Hsu and Rodrigo Girão Serrão. 2023. U-net CNN in APL: Exploring zero-framework, zero-library machine learning. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3589246.3595371

1 Introduction

Specialized machine learning frameworks dominate present deep learning applications. A wide number of highly specialized, optimized libraries exist. These systems are more complex than typical libraries and are better thought of as domain-specific languages (DSLs). While these libraries bolster the current machine learning explosion, because of their highly specialized nature, users tend to become specialists around a very specific framework for computation. This creates a sharp wall that impedes skills transference, where experts can use frameworks effectively but may be underprepared to handle situations that benefit from broader or more flexible skill-sets.[1] This specialization makes sense in the context of traditional programming, where simple networks are accessible, but tedious, while non-trivial ML architectures quickly strain the performance and usability of a "from scratch" implementation. This creates a sharp contrast between simple systems programmed "by hand" and opaque, complex frameworks that work as inflexible black boxes. A lack of profound and intuitive understanding of the underlying mechanics of deep learning systems fosters a type of "programming by knob turning" where networks are programmed via trial and error rather than intentionality, further exacerbating the danger of unintended consequences, already an inherent difficulty in statistical ML [5]. Machine learning frameworks are often highly vendor-specific; even vendor-neutral systems encode greater specificity than warranted for long-running code. This demands higher levels of programmer investment in system maintenance over a product lifetime. Despite this, specialist frameworks prove highly effective, in part because of the performance demands of ML. However, in recent years, reemerging general-purpose array programming languages have presented an alternative potential direction. Such languages

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

were popular during early neural network programming in the 20th century [1], but performance of then-current hardware prevented further development.

APL, a general-purpose array programming language, created by Kenneth Iverson as an improved mathematical notation [13], has risen 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. Recently, new research into APL as a machine learning language has begun. APL's origins in education and its reputation for direct algorithmic expression [16, 17] address some of the above concerns around general-purpose languages and ML. As a linguistically stable language that is both high performance and high level [9], it is suitable for long lived code as well as rapid prototyping, and the data-parallel by default semantics suggests obvious applications to GPU programming. While these advantages might recommend APL for machine learning, the majority of implementations have run on the CPU only, usually via interpreter. While challenging to compile, recent innovations to the language (particularly those with a functional programming focus) make compilation more tractable, and the Co-dfns compiler now exists as an APL implementation with native GPU support [9].

Given the available APL technology and the paucity of existing research on modern ML in APL, we explored APL as a ML language in terms of language design and runtime performance by implementing and benchmarking the U-net convolutional neural network [25]. 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, non-trivial architecture. This makes an ideal candidate for exploring APL's capabilities.

We make the following contributions:

- A from scratch implementation of the U-net convolutional neural network in APL
- Our U-net implementation is exceptionally simple, concise, transparent, and direct
- Our implementation consists of pure APL and no dependencies, frameworks, libraries, or other supporting code
- 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

2 Background

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 popularized with the publication of a paper [18] that used CNNs to achieve state-of-the-art performance in labeling pictures of the ImageNet [4] challenge. However, a prominent paper from 1998 [19] 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 [23]. 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 behavior of a convolutional layer depends on layer parameters like the padding and the stride used [6].

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 neighborhoods [26] by aggregating low-level information to recognize larger features of interest, with the rationale that typical image features (e.g., the recognition or segmentation of objects, or image labeling) consist of regions of pixels and not individual pixels.

2.2 Original U-net Architecture

Ronneberger et al. [25] introduced the U-net architecture, a novel CNN architecture that won several biomedical image segmentation challenges. Since the original implementation in Caffe [14], others have implemented U-net in numerous other frameworks and systems[3], including PyTorch [24].

Figure 1 shows the original diagram that represents the U-net architecture [25]. The blue right arrows, labeled "conv 3x3, ReLU", represent unpadded convolutions with 3×3 kernels. After each of these convolutions, the size of the feature map decreases by 2, from which it can be inferred that the stride [6] is 1. A rectified linear unit (ReLU) [22] activation function follows each convolution. After two such convolutions, max-pooling operations represented by the red down arrows downsample each feature map to half the size using a 2×2 region with stride 2.[4] After every downsampling step, the first convolution doubles the number of channels. The

[2]"channel" typically refers to the leading dimension of these inputs/outputs, a nomenclature that is derived from the fact that CNNs were popularized in the context of image processing.
[3]Numbers by Papers with Code as of March, 2022.
[4]This makes the input size somewhat brittle. Specifically, the input dimensions must be congruent to $12 \mod 16$

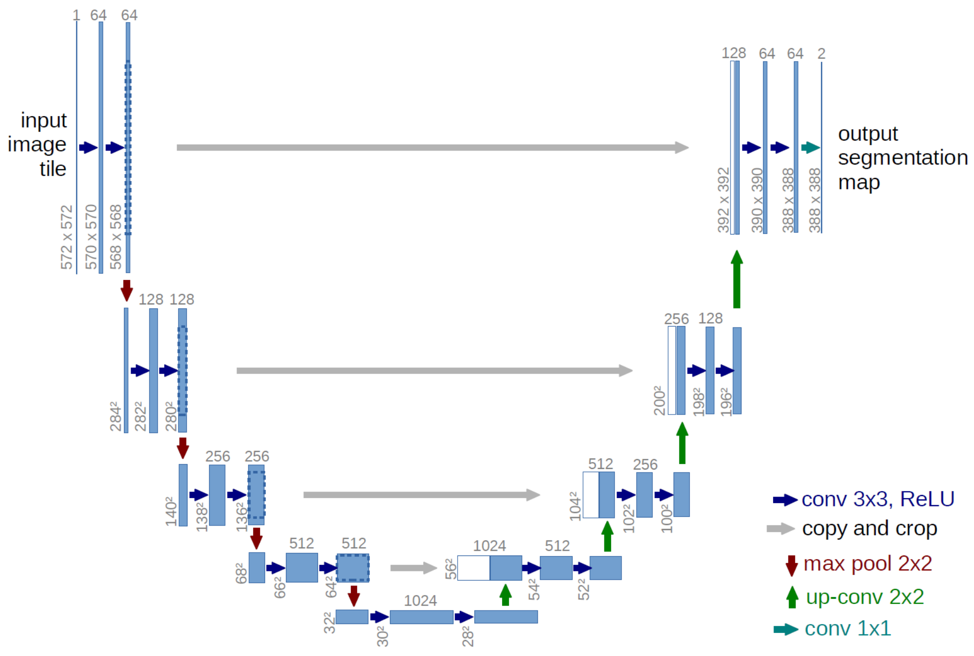

Figure 1. Original u-net architecture, as seen in the original paper [25]. Arrows represent operations between the multi-channel feature maps represented by the rectangles. The number on top of each rectangle is its number of channels and the numbers in the lower-left corner are the x and y dimensions of the feature maps.

pattern of two convolutions (with ReLUs) followed by down-sampling via max-pooling happens four times and makes up the contracting path of the network, on the left of the diagram.

After the contracting path (at the diagram bottom) completes, a corresponding expanding path reduces the channels while increasing the map size. Expansion uses unpadded 3×3 convolutions preceded by an upsampling operation (green up arrows) that doubles the map size while halving the channels. We infer that the original authors used a transposed convolution (with 2×2 kernels) of stride 2 [6][5]. Half of the channels to the initial convolution after upsampling come from the corresponding map on the contracting side of the architecture, as represented by the gray right long arrows in the middle of the diagram. The copied maps are center cropped to ensure matching dimensions. At the end of the contracting path, a 1×1 unpadded convolution reduces the 64 feature maps to 2 feature maps (one per class).

2.3 APL Notation

APL, introduced by Turing award winner Kenneth E. Iverson in the '60s [13], is an executable mathematical notation [12]. This section introduces the basics of the notation used in this paper, but Legrand [20] more fully describes the language. Online interactive systems allow for experimentation with the language without needing to install anything.[6] All notation used here is compatible with Dyalog APL 18.0[7].

[5]See [27] for an informal discussion of this inference.

Table 1. Array names according to *rank*.

Rank	*Name*
0	Scalar
1	Vector
2	Matrix
3	Cuboid
4	Hypercube
5+	Noble

2.3.1 Arrays.

The primary datatype of an APL system is the array.[8] In APL, an array is an N-dimensional rectangular arrangement of one or more elements in row-major order according to a list of zero or more orthogonal dimensions (axes) called its SHAPE, with the number of axes called its RANK. Table 1 gives the common names for sub-classes of arrays by rank. Scalars have exactly one element. Higher ranked, non-empty arrays contain as many elements as the product of its axes, while those with at least one zero length

[6]https://tryapl.org
[7]https://dyalog.com/
[8]Other types may exist for commercial systems, but these are not treated here.

axis, called EMPTY arrays, have a single element called the PROTOTYPE, or FILL, element. For all numeric arrays, such as those used in this paper, the fill element is 0, and it is used by some primitives to extend or resize arrays to larger shapes. Indices begin at 0, rather than 1.[9]

The elements of an array consist solely of other arrays, allowing APL to have a single, uniform datatype.[10] To make computation meaningful, APL defines a set of scalar arrays that correspond to primitive values in the system, such as numbers and characters. Thus, when we write a number like `1`, `3.5`, or `1e¯8`, this is the scalar array corresponding to that number in the APL system.

2.3.2 Expressions. Expressions consist of a series of literals/variables, applications, bindings, or assignments. Literal expressions describe an array. A single numeric value is a literal. Adjacent values, separated by whitespace, form a vector of those values. For example:

```
¯5    ⍝ Scalar negative 5
5 3 1 ⍝ Vector of 3 elements
```

Note the use of a high minus to indicate a literal negative number instead of `-5`, which is the application of the negate function to `5`.

Functions can be first or second order. First order functions, called FUNCTIONS, take arrays as arguments and return a single array value as a result, while second order functions, called OPERATORS, take arrays or FUNCTIONS as arguments and return a FUNCTION result. From now on, we use the term FUNCTION to refer to first-order functions only. Application takes one of the following infix forms:

```
   fn ⍵  ⍝ Monadic function application
 ⍺ fn ⍵  ⍝ Dyadic  function application
⍺⍺ op    ⍝ Monadic operator application
⍺⍺ op ⍵⍵ ⍝ Dyadic  operator application
```

The terms MONADIC and DYADIC refer to arity, not monads in the functional programming sense. We use `⍺`, `⍵`, `⍺⍺`, and `⍵⍵` to refer to the left and right arguments (for function application) and operands (for operator application), respectively. Operators have higher binding precedence than functions, and operators always associate to the left while functions always associate to the right. This applies even in cases where traditional math primitives are used. Consider the following example, where `⌿` is a monadic operator, and `.` a dyadic:

```
   +.×⌿-2×3-4+3×5
←→ (((+.×)⌿)(-(2×(3-(4+(3×5))))))
```

Notice that `-` is applied both monadically and dyadically.

2.3.3 Primitives. APL is known for its primitives: built-in functions and operators assigned special symbols instead of the standard alphanumeric variable names given to user-defined values. The examples used in this paper are expressions of mostly primitive functions and operators.

The largest class of primitives are the SCALAR FUNCTIONS, defined over a single scalar value or values and extended to all arrays by point-wise, also called element-wise, extension. The left and right arguments of a scalar function must share the same shape, with scalar arguments extended to the same shape as the other argument. Consider the following example using addition:

```
      5+1 3 7 10 ⍝ Scalar extension
6 8 12 15
```

Scalar functions return arrays the same shape as their inputs. The set of scalar functions used in this paper include the standard arithmetic functions `+`, `×`, `÷`, `-`, `⌈` (ceiling or maximum), `⌊` (floor/minimum), `|` (absolute value), and `*` (exponent), as well as the Boolean relations `<`, `≤`, `=`, `≠`, `≥`, `>`, and `~` (not) which return `1` for TRUE and `0` for FALSE.

The core primitive `⍴` is used for the shape of an array. The monadic application `⍴⍵` returns a vector of the axes of `⍵`. The dyadic application `⍺⍴⍵` returns an array whose shape is `⍺` and whose elements are taken in wrapping order from `⍵`. For example:

```
      3 5⍴1 2 3 4 ⍝ A 3×5 matrix
1 2 3 4 1
2 3 4 1 2
3 4 1 2 3
```

The function `≢⍵` applied monadically returns the length of the first axis of `⍵`, that is, the first element of `⍴⍵`.

The primitives `⊖` and `⌽` rotate arrays along the first and last axis, respectively. Applied monadically, they reverse/flip an array along the appropriate axis. When applied dyadically, they shift an array by the specified amount. Thus `¯2⌽⍵` rotates (with wrapping) `⍵` two positions to the right along the last axis. Writing `⌽[x]` or `⊖[x]` specifies rotation along axis `x`.

```
      ¯1⊖3 5⍴1 2 3 4 ⍝ Shift down 1 row
3 4 1 2 3
1 2 3 4 1
2 3 4 1 2
```

The transpose primitive `⍉` permutes its argument's axes into different positions. Monadic `⍉⍵` reverses the shape of `⍵`, while giving `⍉` a left argument indicates how to permute the dimensions. Thus, `0 1 3 2⍉⍵` transposes a hypercube `⍵` by exchanging the last two axes between themselves; likewise, `3 0 1 2⍉⍵` transposes `⍵` so that its first axis becomes its last, while all other dimensions retain the same relative order. Consider the following example of transposing two matrix slices (sub-arrays) of a cube:

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

[9]Dyalog APL defaults to 1-based indexing.

[10]This historically contentious design choice will be taken for granted. Industrious readers may explore alternative designs elsewhere.

```
1 2 3

4 0 1
2 3 4
0 1 2
      0 2 1⍉2 3 3⍴0 1 2 3 4
0 3 1
1 4 2
2 0 3

4 2 0
0 3 1
1 4 2
```

The functions ↑ and ↓ resize ⍵ by taking or dropping elements. The expression `2 ¯3↑⍵` will give a `2×3` matrix from ⍵ with rows taken from the top and columns taken from the right, since a negative size takes from the tail instead of the head. Drop (↓) works the same way, but eliminates instead of taking items. The following example centers a `2×2` matrix into a `4×4` matrix. Notice how the matrix is extended using `0` as the FILL element.

```
      2 2⍴1 2 3 4
1 2
3 4
      3 3↑2 2⍴1 2 3 4
1 2 0
3 4 0
0 0 0
      ¯4 ¯4↑3 3↑2 2⍴1 2 3 4
0 0 0 0
0 1 2 0
0 3 4 0
0 0 0 0
```

The expression `⍺,⍵` catenates ⍺ and ⍵ together as a single array along the last axis. However, in this paper we primarily use `,` monadically. The expression `,⍵` gives a vector of the elements of ⍵, collapsing the axes of ⍵ into a single axis whose size is the product of the axes of ⍵. We write `,[d]⍵` to collapse the specified axes `d` to a single axis. The following example collapses the trailing two axes of a cube, creating a matrix:

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

4 0 1
2 3 4
0 1 2
      ,[1 2]2 3 3⍴0 1 2 3 4
0 1 2 3 4 0 1 2 3
4 0 1 2 3 4 0 1 2
```

We write `N⌿⍵` or `N/⍵` to mean the array ⍵ with each element duplicated `N` times along the first or last axis, respectively, as follows:

```
      2 2⍴1 2 3 4
1 2
3 4
      2⌿2/2 2⍴1 2 3 4
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
```

The expressions `f⌿⍵` and `f/⍵` compute the reduction of ⍵ using function `f` over the first or last axis, respectively. This is often called a right fold in many functional languages.

```
      +⌿1 2 3 4 5
15
```

The dyadic application `+.×` is the function for matrix multiplication extended to higher dimensions.

We can mutate the elements of an array or bind the result of an expression to a name. The form `V←expr` binds `V` to the result of `expr`. The expression `A[i]←expr` assigns the elements of `expr` to the corresponding elements of `A` according to indices `i`. If we have a nested array `A` in which we want to store a single non-scalar array `X` at index `i`, we box the value `X` into a scalar first, using `⊂`, resulting in the expression `A[i]←⊂X`. We write `i⊃A` to extract the same array `X` from `A`.

The syntax `{expr+ ...}` returns a function. Within the body of the function, consisting of a set of expressions, ⍺ is the left argument to the function and ⍵ is the right, while ∇ refers to the function itself, permitting anonymous recursion. Functions execute expressions in order and return the first unbound expression or the last bound expression.

```
      add1←{1+⍵}
      add1 3 4 5
4 5 6
```

3 Implementation

3.1 Overview

Our implementation of U-net divides 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 leverage features of APL to implement both aspects of the system, and so we treat each in its own sub-section.

Because Co-dfns does not yet support the complete array of Dyalog primitives and their semantics, we could enhance some implementation techniques through the use of the richer feature-set. The effect of these features would be increased concision and clarity, but such improvements would not significantly affect the overall performance of the code, either positively or negatively. The overall structure of the code is clear and simple enough in its current state to warrant inclusion almost verbatim, but some adjustments were made to reduce the total number of primitives used, which makes the code slightly less elegant while reducing the burden to the reader to learn more APL primitives.

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

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 [10], stands out as the most obviously aligned with convolutional neural networks. The J programming language introduced an alternative implementation of the stencil operator earlier [15], from which Dyalog derived inspiration for the implementation of their own stencil operator. We propose an alternative that was first suggested to us by the late Roger Hui, the stencil FUNCTION [11]. 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 (padded) of the stencil function using the Dyalog stencil operator might be:

```
SF←{({⍵}⌺⍺)⍵}
```

A naïve implementation of the stencil function that did not pad its results was implemented in Co-dfns and used in U-net implementation. We subsequently use `⌺` to mean the stencil FUNCTION without padding and not the stencil OPERATOR as it appears in Dyalog APL. Given a specification `s` as the left argument, the stencil function, written `s⌺a`, returns an array of each sliding window slice of `a` specified by `s`. The first row of the specification indicates the window sizes while the second row indicates the step size, or 1 if omitted. The window is conceptually "slid" along the leading axes of the input array. The two most common sliding window sizes for `⌺` in U-net are `3 3` for convolutions, corresponding to a window size of `3×3` and a step of `1` for each axis, and `2 2⍴2` (a 2 by 2 matrix of 2's) for the max pooling layers and up convolutions, corresponding to a `2×2` window size and a step of `2` for each axis.

For example, a `3 3` sliding window of step 1 on a `6 6` matrix returns an array of shape `4 4 3 3` consisting of 16 `3 3` matrix slices arranged in a `4 4` grid. The dimensions of the two leading axes will shrink by 2 each time because we implemented stencil without padding.

```
      6 6⍴⍳36
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35
      ⍴3 3⌺6 6⍴⍳36
4 4 3 3
```

In the following, we box each `3 3` matrix slice using `⊂[2 3]` to show the slices in a more compact form as a nested matrix of matrices, but the result of `⌺` is actually a rank 4 hypercube in this case.

```
      ⊂[2 3]3 3⌺6 6⍴⍳36
┌────────┬────────┬────────┬────────┐
│ 0  1  2│ 1  2  3│ 2  3  4│ 3  4  5│
│ 6  7  8│ 7  8  9│ 8  9 10│ 9 10 11│
│12 13 14│13 14 15│14 15 16│15 16 17│
├────────┼────────┼────────┼────────┤
│ 6  7  8│ 7  8  9│ 8  9 10│ 9 10 11│
│12 13 14│13 14 15│14 15 16│15 16 17│
│18 19 20│19 20 21│20 21 22│21 22 23│
├────────┼────────┼────────┼────────┤
│12 13 14│13 14 15│14 15 16│15 16 17│
│18 19 20│19 20 21│20 21 22│21 22 23│
│24 25 26│25 26 27│26 27 28│27 28 29│
├────────┼────────┼────────┼────────┤
│18 19 20│19 20 21│20 21 22│21 22 23│
│24 25 26│25 26 27│26 27 28│27 28 29│
│30 31 32│31 32 33│32 33 34│33 34 35│
└────────┴────────┴────────┴────────┘
```

Notice that the window slides along the leading axes, so each slice captures all trailing axes for each slice.

3.3 Neural Network Vocabulary

The original U-net paper uses five distinct operations to describe the network (see Figure 1 on page 3):

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.

Each of our operations works over a given network “layer,” which is a cube of shape `N M C` where `N M` are the layer dimensions and `C` is the number of layer channels.

3.3.1 Convolution (3×3) with ReLU.

The primary U-net convolutional kernel 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 kernel `α` of shape `3 3 I O` of `I` input channels and `O` output channels with a layer `ω` of shape `N M I` that obeys the following shape invariant:

```
ρα CV ω ←→ (¯2+2↑ρω),¯1↑ρα
```

This says that the size of the first two axes of `α CV ω` will be two less than the first two axes of `ω`, while the remaining axis, corresponding to the output channels, will match the last axis of `α`. In other words, given the kernel of shape `3 3 I O` and layer of shape `N M I`, the result of `α CV ω` will have 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←{0⌈(,[2 3 4]3 3⌺ω)+.×,[0 1 2]α}
```

The result of `3 3⌺ω` will have the shape `(N-2) (M-2) 3 3 I`, so the expression `,[2 3 4]3 3⌺ω` gives us a cube of shape `(N-2) (M-2) (3×3×I)` while the expression `,[0 1 2]α` gives us a matrix of shape `(3×3×I) O`. We use the extended matrix multiplication `+.×` to compute the convolution, resulting in a cube of shape `N M O`. The ReLU function `0⌈ω` follows as the final operation.

Computing 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:

```
ΔCV←{
      w←⊖⌽[1]0 1 3 2⍉α
     Δz←ω×0<a
     Δw←3 0 1 2⍉(⍉,[0 1]Δz)+.×,[0 1]3 3⌺x
     ΔZ←¯2⊖¯2⌽[1](4+2↑ρΔz)↑Δz
     Δx←(,[2 3 4]3 3⌺ΔZ)+.×,[0 1 2]w
}
```

Since our stencil function does not pad its results, the expression `¯2⊖¯2⌽[1](4+2↑ρΔz)↑Δz` expands the shape of `Δz` to ensure that the output convolution dimensions are 2 greater than those of `Δz`.

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 `↓`:

```
CC←{
     p←((ρα)-ρω)÷2),ω
     ((⌊p)↓(-⌈p)↓α
}
```

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 3.4.

```
ΔCC←{
     n m←-⌊(2↑(ρα)-ρω)÷2
     Δx←n⊖m⌽[1](ρα)↑ω
}
```

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 definition for `MX`:

```
MX←{⌈⌿[2],[2 3](2 2ρ2)⌺ω}
```

Here we write `⌈⌿[2],[2 3]ω` 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 2ρ2)⌺ω` is a rank 5 array of shape `(N÷2) (M÷2) 2 2 C`. We then collapse axes 2 and 3 to form an array of shape `(N÷2) (M÷2) 4 C` and subsequently find the maximum value for each vector along axis 2, resulting in an array of shape `(N÷2) (M÷2) C`.

Computing the backpropagation of this involves replicating each of the stencil dimensions, which are the two leading axes in our implementation. Given an input `α` and output layer `ω` the following function `ΔMX` computes the backpropagation:

```
ΔMX←{
     y←(ρα)↑2⌿2/[1]ω
     y×α=y
}
```

3.3.4 Transposed Convolution (2×2).

The upsampling computation with convolution proved to be the most subtle and challenging, mostly due 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 hits that an upsampling convolution matches the shape and form of a non-overlapping sliding window in the output layer, the computation becomes much clearer.[11]

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←{,[0 1],[2 3]0 2 1 3 4⍉ω+.×α}
```

The key change here from the reliance on `+.×` with `CV` is the use of a dyadic transpose. The input into the `⍉` transpose is the result of `ω+.×α` which has shape `N M 2 2 O`. Thus, the expression `0 2 1 3 4⍉ω+.×α` returns an array 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×2) (M×2) O`.

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

```
ΔUP←{
    Δw←(⍉,[0 1]x)+.×,[0 1](2 2ρ2)⌺Δz
    Δx←(,[2 3 4](2 2ρ2)⌺Δz)+.×⍉,[1 2 3]α
}
```

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` of `ω+.×α`, which we combine with the soft-max layer described in the paper:

```
C1←{
    z←ω+.×α
    z←*z-[0 1]⌈/z
    1E¯8+z÷[0 1] +/z
}
```

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

```
ΔC1←{
    Δw←(⍉,[0 1]x)+.×,[0 1]Δz
    Δx←Δz+.×⍉w
}
```

3.4 U-net Architecture

Given the core vocabularies defined in 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 on page 3. To do this, we observe that the structure of the U-net diagram is an almost symmetric pattern. The output layer computations form three 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 on page 10 contains a visual arrangement of the kernel shapes used in our architecture mirroring the overall structure of Figure 1 on page 3.

Additionally, we note that the U-shaped structure also mimics 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 recurring 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 `α+i` to access the *i*th column of the network described in Table 2 on page 10 at the depth `α÷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)ρ⊂θ
  ⍝ 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 (3↑(ρω),1)ρω
}
```

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×(ρω)ρα⊃V
    W[α]←⊂(α⊃W)-LR×α⊃V
  }
  ⍝ Back propagation layers ...
  ΔLA←{
```

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

```
    α≥≢Z:ω
    up←(α+5)ΔUP(-(≢⍉ω)÷2)↑[2]ω
    down←(α+6)∇(α+3)ΔCV(α+4)ΔCV up
    mx←(α+2)ΔMX down
    (α+0)ΔCV(α+1)ΔCV mx+(α+2)ΔCC ω
  }
  diff←YΔ-(~Y),[1.5]Y
  3 ΔLA 0 ΔCV 1 ΔCV 2 ΔC1 diff
}
```

4 Performance

We primarily focused on comparing our U-net implementation against a reference implemented in PyTorch [24], which is an easy to use Python framework with good 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 on page 10. 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 NVIDIA 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 [3, 25]. 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 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.

The stencil function was modeled in APL and used to conduct the above benchmark. The model, written in APL, is a naïve 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 naïve 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.

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

	Operation					
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
	Downward Pass			*Upward Pass*		

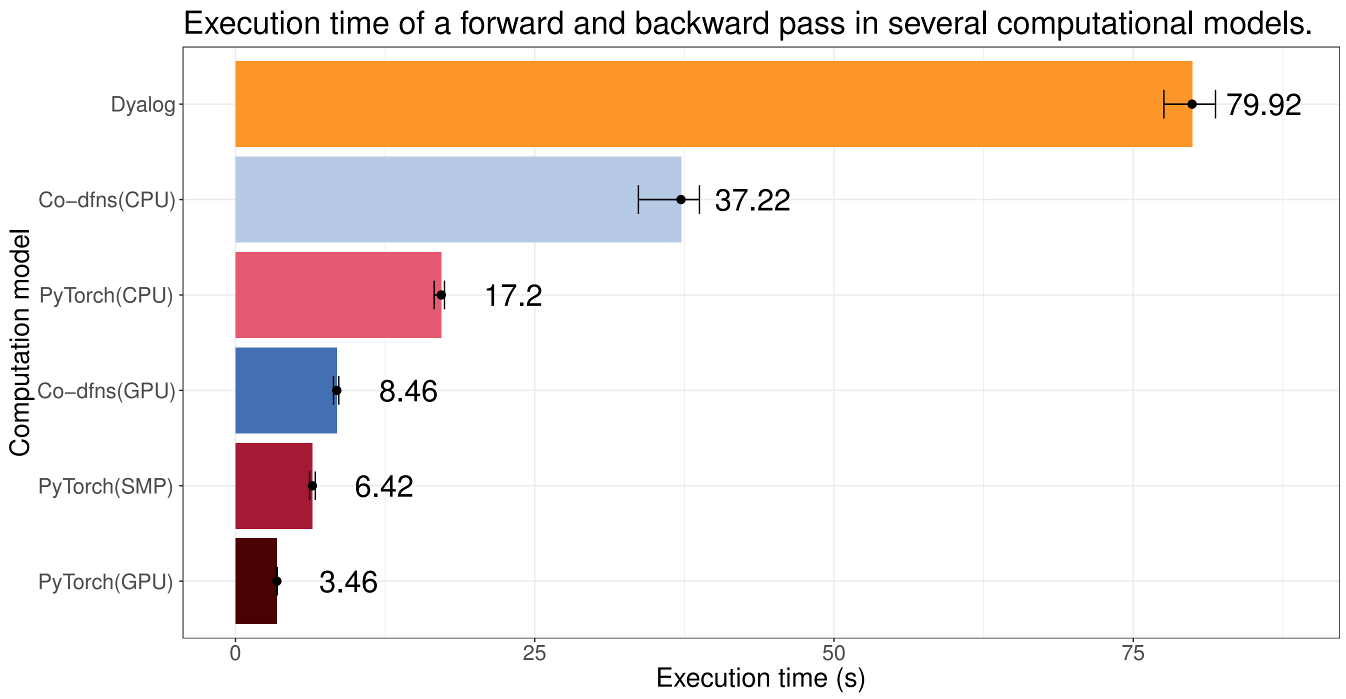

Figure 2. Performance results for U-net across a range of platforms

5 Discussion

5.1 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 transferable 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.

5.2 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 foreknowledge around their chosen language implementation.

It remains unclear, however, whether, if this foreknowledge 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 skill-set 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

In addition to the Co-dfns compiler, Šinkarovs et al. [28] have explored alternative implementations to CNNs, though not U-net specifically. They focused on APL as a productivity enhancement for CNN development, and only benchmarked the APL implementation on the CPU using the Dyalog APL interpreter. They make a case for the usability of APL even with the performance numbers they achieved. Research in SaC demonstrates implementations competitive with the current state of the art [31]. 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 [8], 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. Futhark has been used to implement deep learning primitives [29], and 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 [21], Julia [2], and Python/ Numpy [7, 30], 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 performance potential of the specialized convolution functions against our naïve implementation when using the same backend in Co-dfns; initial tests showed worse results and were omitted here.

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 naïve nature of our implementation and the naïve 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

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

[2] 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.

[3] 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

[4] 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.

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

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

[7] 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

[8] 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.

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

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

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

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

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

[14] 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).

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

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

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

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

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

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

[21] Inc Math Works. 1992. *MATLAB reference guide*. Math Works, Incorporated.

[22] 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).

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

[24] 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

[25] 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.

[26] 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.

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

[28] 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.

[29] Duc Minh Tran, Troels Henriksen, and Martin Elsman. 2019. Compositional Deep Learning in Futhark. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Berlin, Germany) *(FHPNC 2019)*. Association for Computing Machinery, New York, NY, USA, 47–59. https://doi.org/10.1145/3331553.3342617

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

[31] Artjoms Šinkarovs, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2021. Array Languages Make Neural Networks Fast. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/3460944.3464312

Appendix A: Complete APL U-net implementation

```
:Namespace UNET

 W←⍬ ⋄ V←⍬ ⋄ Z←⍬ ⋄ LR←1e¯9 ⋄ MO←0.99

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

 BCK←{Y←⍺ ⋄ Y∆←⍵
  ∆←{0⊣W[⍺]←⊂(⍺⊃W)-LR×⊃V[⍺]←⊂⍵+MO×(⍴⍵)⍴⍺⊃V}
  ∆CV←{w←,[⍳3]⊖⌽[1]0 1 3 2⍉⍺⊃W ⋄ x←⊃⍺⊃Z ⋄ ∆z←⍵×0<1⊃⍺⊃Z
   ∆Z←¯2⊖¯2⌽[1](4+2↑⍴∆z)↑∆z
   _←⍺ ∆ 3 0 1 2⍉(⍉,[⍳2]∆z)+.×,[⍳2]3 3⌺x
   w+.×⍨,[2+⍳3]3 3⌺∆Z}
  ∆CC←{x←⍺⊃Z ⋄ ∆z←⍵ ⋄ d←-⌊2÷⍨2↑(⍴x)-⍴∆z ⋄ (⊃d)⊖(1⊃d)⌽[1](⍴x)↑∆z}
  ∆MX←{x←⍺⊃Z ⋄ ∆z←⍵ ⋄ y×x=y←(⍴x)↑2⌿2/[1]∆z}
  ∆UP←{w←⍺⊃W ⋄ x←⍺⊃Z ⋄ ∆z←⍵ ⋄ cz←(2 2⍴2)⌺∆z
   _←⍺ ∆(⍉,[⍳2]x)+.×,[⍳2]cz
   (,[2+⍳3]cz)+.×⍉⍪w}
  ∆C1←{w←⍺⊃W ⋄ x←⍺⊃Z ⋄ ∆z←⍵ ⋄ _←⍺ ∆(⍉,[⍳2]x)+.×,[⍳2]∆z ⋄ ∆z+.×⍉w}
  ∆LA←{⍺≥≢Z:⍵
   down←(⍺+6)∇(⍺+3)∆CV(⍺+4)∆CV(⍺+5)∆UP ⍵↑[2]⍨-2÷⍨⊃⌽⍴⍵
   (⍺+0)∆CV(⍺+1)∆CV(⍵ ∆CC⍨⍺+2)+(⍺+2)∆MX down}
  3 ∆LA 0 ∆CV 1 ∆CV 2 ∆C1 Y∆-(~Y),[1.5]Y}

 E←{-+⌿,⍟(⍺×⍵[;;1])+(~⍺)×⍵[;;0]}

 RUN←{Y Y∆(Y E Y∆)⊣(Y←⌊0.5+nm↑⍵↓⍨2÷⍨(⍴⍵)-nm←2↑⍴Y∆)BCK⊢Y∆←FWD ⍺}

: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)
```

Appendix C: Raw Performance Timings (Secs)

	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

Received 2023-03-31; accepted 2023-04-21