Convolutional Neural Networks in APL

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Robert Bernecky
Snake Island Research Inc
Toronto, Ontario, Canada
bernecky@snakeisland.com

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

Abstract

This paper shows how a Convolutional Neural Network (CNN) can be implemented in APL. Its first-class array support ideally fits that domain, and the operations of APL facilitate rapid and concise creation of generically reusable building blocks. For our example, only ten blocks are needed, and they can be expressed as ten lines of native APL. All these blocks are purely functional, and they are built out of a small number of builtin operators, resulting in a highly portable specification that is immediately runnable and should be suitable for high-performance optimizations and parallel execution. This implies that APL can be seen as a framework to define shallowly-embedded machine learning DSLs without any external dependencies, making them useful at least for experiments and prototyping. We explain the construction of each CNN building block, and briefly discuss the performance of the resulting specification.

CCS Concepts • **Software and its engineering** → **Concurrent programming languages**; **Data types and structures**; • **Theory of computation** → *Design and analysis of algorithms.*

Keywords APL, arrays, neural networks, DSL, CNN

ACM Reference Format:
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 (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3315454.3329960

1 Introduction

Array languages in general, and APL in particular, are often acclaimed to be powerful tools of thought, as suggested in Kenneth E. Iverson's 1979 Turing Award Lecture [9]. APL's design, based on fundamental concepts of natural language, where data are *nouns* and operators such as multiply and iterate are *verbs* and *conjunctions*[1], in combination with a language nucleus of a few pre-defined operators, makes APL an easily accessible tool for domain experts, without requiring a strong background in programming.

Problems requiring handling large amounts of data with some structural regularity are often ideally suited for quick programmatic solutions in APL. One such area that has recently gained significant interest is machine learning and neural networks. Neural networks can be viewed as nested linear approximations of unknown functions that are poorly understood analytically. Calibrating these functions from given input-output pairs, also referred to as "training", boils down to iterating mathematically simple operations over large amounts of data.

While these operations should be ideally suited for array languages such as APL, machine learning experts tend to use several domain-specific frameworks, such as TensorFlow or PyTorch, instead. While there is no doubt that these frameworks are convenient to use, any non-trivial extensions beyond the provided functionality will require understanding the underlying codebase which is difficult due to its enormous size and complex design. This makes domain experts dependant on framework specialists.

Following the APL as "tool for thought" idea, we investigate how difficult would it be to define a sufficient number of TensorFlow-like operators, with a clear mathematical structure, in native APL, to build a state of the art hand-written image recognition CNN [14, 27]. First of all, to our knowledge this has not been done before. Secondly, if the operators were simple to define, we can ask ourselves, whether APL is a reasonable language to host a shallowly-embedded DSL for machine learning.

APL offers a number of purely functional built-in array operations[2]: they have no state and operate on immutable data. New verbs and conjunctions can be expressed as compositions of the extant ones, built-in or user-defined, producing concise, purely functional data-parallel specifications that are immediately executable.

Our study shows that, for our CNN, the requisite operations can be expressed in ten lines of native APL code, using just 22 built-in verbs and conjunctions:

```
blog←{α×ω×1-ω}
```

[1]See https://www.jsoftware.com/papers/APLDictionary.htm for more details.
[2]Although APL includes a number of imperative constructs, we restricted ourselves to a functional subset.

```
backbias←{+/,ω}
logistic←{÷1+*-ω}
maxpos←{(,ω)⍳⌈/,ω}
backavgpool←{2⌿2/ω÷4}⍤2
meansqerr←{÷∘2+/,(α-ω)*2}
avgpool←{÷∘4{+/,ω}⌺(2 2ρ2)⍤2⊢ω}
conv←{s←1+(ρω)-ρα⋄⊃+/,α×(ιρα){s↑α↓ω}¨⊂ω}
backin←{(d w in)←ω⋄⊃+/,w{(ρin)↑(-ω+ρd)↑α×d}¨ιρw}
multiconv←{(a ws bs)←ω⋄bs{α+ω conv a}⍤(0,(ρρa))⊢ws}
```

Such brevity offers several advantages. First, it captures the computational essence of the underlying problem, and facilitates communication of the algorithm to colleagues. The operators are also rank- and shape-polymorphic, making them reusable in various contexts. By polymorphic, we mean that they will operate, unchanged, on arrays of any rank or shape.

Second, our specification can easily be ported to non-APL contexts. None of the built-in operators are neural-network specific, and all of them have precise, well-defined behaviour.

Finally, the functional nature of our specification, with element-wise data-parallel operations, presents many opportunities for compiler technology to generate efficient, parallel code, for CPU or GPU target systems, with no APL source code changes.

The individual contributions of the paper are:

- Framework-free, concise, executable specification of a CNN in APL
- Design exploration of the CNN and its implementation
- Brief run-time performance evaluation

The rest of the paper is organised as follow. We briefly introduce CNNs and APL in Section 2. We explore design trade-offs of the new operators and how to combine them in Section 3. We talk about performance considerations and compiler optimisations in Section 4, discuss related work in Section 5 and conclude in Section 6.

2 Background

2.1 CNN

This section is an overview of machine learning algorithms, focusing on the computational aspects of CNNs. For an in-depth review, refer to [8, 16].

Machine learning algorithms are based on the idea that we want to learn (guess) a function f that maps the input variables X to the output variables Y, *i.e.* $Y = f\ X$, in an optimal way, according to some cost function. After f is found for the existing sample, we want to use f to make predictions for new inputs.

CNNs belong to the class of machine learning algorithms called neural networks. They have a distinctive feature: the function $f : X \rightarrow Y$ that we want to learn is a composition of functions g_i, that can be further decomposed into smaller functions. Overall, such a composition forms a graph (or *network*) connecting inputs X with outputs Y.

A typical function composition takes the form: $f\ x = A\ (\sum_i w_i\ (g_i\ x))$ where A is an activation function (usually chosen to be continuous and differentiable, *e.g.* sigmoid, hyperbolic tangent, *etc.*) and w_i are so-called weights. These weights are parameters of the approximation that we want to find, chosen so as to minimise our cost function.

Usually, neural networks are designed to allow slicing of the elementary functions g_i into layers, so that all elements of a given layer can be computed independently. Layering has the beneficial effect of making that computation highly parallel. A layer is an activation function of the weighted sum of other layers, so most of the transitions in the network can be expressed as matrix or tensor operations.

Very often, due to the network size and complexity, a closed solution that finds optimal weights either does not exist or is very difficult to find. Therefore, weight prediction is usually performed in an iterative manner. In this case, the concept of backpropagation — a method to calculate the gradient of the objective function with respect to the weights, is of significant importance. It provides a working solution that is straight-forward to compute: $w := w - \eta \nabla F(w)$ where w are all the weights in the given network. For the cases in which our objective function can be written as: $F = \sum_i F_i$, the gradient descent can be rewritten as: $w - \eta \nabla \sum_i F_i = w - \eta \sum_i \nabla F_i$. Furthermore, the stochastic gradient descent [26] approximates the true gradient as follows: $w := w - \eta \nabla F_i(w)$ which is typically more efficient. Intuitively, if we process a batch of items, we can update weights after processing one individual item. Finally, with carefully chosen activation functions A, the computation of backpropagation can be expressed as a composition of linear-algebraic operations.

2.2 APL

The design of APL, a language with first-class arrays, is driven by the desire to find an consistent, terse, and efficient notation to communicate mathematical ideas. Although this might seem a purely syntactical consideration, it is taken seriously by advocates of array languages. The emphasis is that the user should express what we want to compute, rather than how it should be computed, using the smallest number of generic, easily composable primitives. The shorter the program is, the quicker we, and others, can understand what it is doing. Furthermore, code that is not there can not break, so shorter programs are inherently more bug-free than longer ones. We offer a bit of APL knowledge here, to ease reading the paper. For the full language reference manual, see [6]. We restrict ourselves to a purely functional subset of APL, and use no explicit array indexing, which simplifies automatic or manual analysis of operation compositions.

Arrays The principal data type in APL is a rectangular n-dimensional array — an object that can be indexed with n-element tuples of natural numbers. Each array carries its shape — an n-element tuple of natural numbers that defines the valid set of indices into that array. For example:

```
      w
0 1 2
3 4 5
6 7 8
```

In this paper, an indented expression is the one we wish to be evaluated, and the non-indented text comprises the result of the computation. In this example, `w` is a variable that evaluates to a 3×3 two-dimensional array with elements $0, 1, 2, \ldots$. We can select elements from such an array with two-element tuples, where both of the elements are less than 3.

We can find the *shape* of an array using the `ρ` function:

```
      ρw
3 3
```

The *rank*, or dimension, of an array can be found from the shape of the shape:

```
      ρρw
2
```

Numbers and characters are zero-dimensional arrays; their shape is an empty vector, and their rank is 0:

```
      ρ5

      ρρ5
0
```

We call 0-dimensional arrays *scalars* and 1-dimensional ones *vectors*.

Arithmetic Functions Functions in APL are of order 1 and 2, and application is right-associative. First-order functions can take either one or two arguments[3]. One-argument ones have a prefix form; two-argument ones an infix one. For example:

```
      ρρw
2
      2×2+2
8
```

Right associativity lets us avoid parentheses in the first expression; we get 8, rather than 6, in the second one. Typical arithmetic operations include: `+ - × ÷`, the `*` stands for power, *i.e.* `2 * 3` evaluates to 8. A sequence of values is automatically merged into an array:

```
      2 3 4 5
2 3 4 5
```

Arithmetic functions are automatically lifted to the element level, when applying to arrays of the same shape:

[3]In APL parlance, one-argument functions are called *monadic* and two-argument ones *dyadic*. In this paper, we call them one- and two-argument.

```
      2 3 4 5 + 2 3 4 5
4 6 8 10
```

When one of the arguments is an array and the other argument is a scalar, the scalar is extended to be an array of the same shape as the other argument: `1 + 2 3 4 5` evaluates to `3 4 5 6` as well as `2 3 4 5 + 1`.

Nested Arrays APL supports nested arrays — those with non-homogeneous elements. This is achieved, primarily, by means of two functions: enclose `⊂` and disclose `⊃`. All elements of APL arrays are zero-dimensional scalars. Enclose creates, from any argument array, a scalar, which can then become an element of other arrays. The value of that argument array can be be retrieved by applying disclose to that scalar. The overall concept is similar to the notion of pointers, in which enclose takes a reference to an object, and disclose dereferences the reference. For example:

```
      ⊂w
┌─────┐
│0 1 2│
│3 4 5│
│6 7 8│
└─────┘
```

The box, a display artifact of the APL session, shows that the value is enclosed. The shape of `⊂w` is the empty vector, hence is a scalar. When non-scalar values are written in sequence, the resulting array contains enclosed elements of that sequence. For example:

```
      (2 3) w
┌───┬─────┐
│2 3│0 1 2│
│   │3 4 5│
│   │6 7 8│
└───┴─────┘
```

The shape of the above array is `2`. In order to access the element `4` in the above array we can select (using the `▯` operator) the second vector element, disclose it, then select from it at index `1 1`:

```
      1 1 ▯ ⊃ 1 ▯ (2 3) w
4
```

Array Functions The one-argument `,` (comma) function flattens any multi-dimensional array, in row-major order. The two-argument version of `,` (comma) concatenates arrays on the last axis, implying that all the shape elements of both arguments, except the last one, must be the same:

```
      (,w) (w,w)
┌─────────────────┬───────────┐
│0 1 2 3 4 5 6 7 8│0 1 2 0 1 2│
│                 │3 4 5 3 4 5│
│                 │6 7 8 6 7 8│
└─────────────────┴───────────┘
```

The two-argument ⍴ function flattens the right argument array, and reshapes it to the shape of its left argument. If the right argument has more elements than is required by the shape, the remaining elements are ignored. If there are fewer elements, array elements are replicated:

```
      (2 2⍴w) (3 3⍴1 2) (9⍴w)
┌───┬─────┬─────────────────┐
│0 1│1 2 1│0 1 2 3 4 5 6 7 8│
│2 3│2 1 2│                 │
│   │1 2 1│                 │
└───┴─────┴─────────────────┘
```

A one-argument version of ⍳ creates an array of the shape provided by its argument, in which a given element contain the value of the index of that element:

```
      (⍳5)(⍳2 2)(⍳2 1 1)
┌─────────┬─────────┬───────┐
│0 1 2 3 4│┌───┬───┐│┌─────┐│
│         ││0 0│0 1│││0 0 0││
│         │├───┼───┤│└─────┘│
│         ││1 0│1 1││┌─────┐│
│         │└───┴───┘││1 0 0││
│         │         │└─────┘│
└─────────┴─────────┴───────┘
```

Higher-order Functions Second-order functions in APL accept first-order ones as their arguments. A simple example is the function composition operator ∘. It acts in the usual way: (f∘g) e is the same as f(g e). Composition can be also used for partial applications of two argument functions. We compose the argument with the function either on the right or on the left:

```
      ((÷∘4) 2) ((⍴∘1) 2 2)
┌───┬───┐
│0.5│1 1│
│   │1 1│
└───┴───┘
```

Composition is often useful because it can save parentheses. Consider a long expression, *e.g.* ,w+3×2, that we are want to divide by four. Instead of writing (,w+3×2)÷4 we can write ÷∘4,w+3×2. Since element concatenation has a higher priority than the function application, the +∘3 4 expression would not work as one might expect. Instead of doing (+∘3) 4 which evaluates to 7, APL does +∘(3 4) which produces a one-argument function that adds 3 4 to its argument. To avoid extra parentheses, it is convenient to use an identity function, such as ⊢, to inhibit implicit concatenation:

```
      +∘3 ⊢ 4
7
```

This has some similarity with the $ operator in Haskell or @@ in Ocaml.

The reduce operator f/ (slash) inserts the two-argument function f among the elements of the argument array, on the last axis:

```
      (+/w) (×/1 2 3) (+/1 2 3)
┌───────┬─┬─┐
│3 12 21│6│6│
└───────┴─┴─┘
```

If the argument array is a single element, then the result is that element. For empty vectors, +/ and ×/ produce 0 and 1 correspondingly, but for an arbitrary function, reducing an empty array is an error.

We use two *element-wise*-like operators: each ¨ and rank ⍤ [2, 3]. The f¨ applies f to each element of the argument array and in case f evaluates to a non-scalar result, a nested array is produced. For example:

```
      (⍳¨2 3) (1∘+¨2 2⍴⍳4)
┌───────────┬───┐
│┌───┬─────┐│1 2│
││0 1│0 1 2│││3 4│
│└───┴─────┘│   │
└───────────┴───┘
```

The rank conjunction operator f⍤n is a second-order function that applies f to the sub-arrays of the n innermost dimensions of the argument array. When results of f applications have the same shape s, but are non-scalar, the shape of the result is the concatenation of the argument array shape and s. For example:

```
      ⍴2 2∘⍴⍤0⊢2 3
2 2 2
```

Here, we apply 2 2∘⍴ to the elements of a two-element vector 2 3, obtaining two 2×2 arrays, called *cells*, that are combined (laminated in APL lingo) into an array of shape 2 2 2. When cell result shapes of f are non-uniform, the largest shape is taken, and all the other arrays are padded with zeroes at the end of each axis. In our examples, this situation does not arise.

Both operators allow f to be a two-argument function, in which case the derived operator requires argument arrays on the right and on the left. For the case of the rank operator, we also have to specify two ranks: for the left and right arguments. For example:

```
      (2 3⍴¨3 4) ((2 2⍴2)⍴⍤(1 0)⊢2 3)
┌───────────┬───┐
│┌───┬─────┐│2 2│
││3 3│4 4 4│││2 2│
│└───┴─────┘│   │
│           │   │
│           │3 3│
│           │3 3│
└───────────┴───┘
```

The second expression is of shape 2 2 2, where the first 2 2 sub-array consists of 2s and the second one of 3s.

Defining functions A function is defined using the `{}` block that surrounds the body text. For example, a constant function that always produces 42 can be defined as `{42}`. Function arguments are bound to variables `α` and `ω` for the left and right arguments, respectively. For example, `{ω + 1} 5` evaluates to `6`. A function can have local variable bindings using the assignment operator `←`. Multiple statements can be separated either by the `⋄` operator or a line break. For example, the following function computes the number of elements in the array:

```
      {s←ρω ⋄ p←×/s ⋄ p} w
9
```

The result of a function is the first statement that is not given a name. Often, as in the above, this is its last statement.

As we have mentioned before, we elided details about operator semantics that are not relevant to understanding the rest of the paper. For precise specification, please refer to [6].

Variable names In the rest of the paper, we use variable names with unicode symbols. For example: `k¹`, `C²`, `δŷ`, *etc.* Even though this is not supported by current APL interpreters, it may help to relate the code with its mathematical formulation, and thereby increase the overall readability.

3 CNN Building Blocks

This paper demonstrates the use of a functional subset of APL to solve a classical hand-written digit recognition problem, using Zhang's algorithm, as informally presented in Fig. 1, with input data from the widely used MNIST data set [4]. The Zhang paper contains a formal mathematical specification of the algorithm; we present that specification as a small set of defined APL functions, striving for a good balance between conciseness and efficiency. The full version of this code is freely available at https://github.com/ashinkarov/cnn-in-apl.

Convolution The first layer C_1, computes six convolutions of the 28×28 input image I with 5×5 matrices of weights $k^1_{1,i}$. Each convolution computes a weighted sum at every position of I where $k^1_{1,i}$ could be placed atop without truncation. This produces six 24×24 arrays.

Consider a single convolution over a 2d image I, with weights w. In order to facilitate such computations, Dyalog APL recently (Version 16) introduced a new operator, *stencil*, denoted as `⌺`, whose semantics prescribe that the array is padded with zeroes, and then a user-defined convolution is applied to a sliding window across the padded array, for example:

```
      {⊂ω}⌺(3 3)⊢3 3ρ1
┌─────┬─────┬─────┐
│0 0 0│0 0 0│0 0 0│
│0 1 1│1 1 1│1 1 0│
│0 1 1│1 1 1│1 1 0│
├─────┼─────┼─────┤
│0 1 1│1 1 1│1 1 0│
│0 1 1│1 1 1│1 1 0│
│0 1 1│1 1 1│1 1 0│
├─────┼─────┼─────┤
│0 1 1│1 1 1│1 1 0│
│0 1 1│1 1 1│1 1 0│
│0 0 0│0 0 0│0 0 0│
└─────┴─────┴─────┘
```

[4]See http://yann.lecun.com/exdb/mnist/.

The algorithm applies stencil to the argument 3×3 array of 1-s `3 3ρ1`, with a sliding window of size `3 3`, computing the enclose `{⊂ω}` of each such subarray. The zeroes at the non-central cells comprise the padding that will be dropped, as it is not relevant to this algorithm, resulting in the following code:

```
      t ← ⌊÷∘2(ρw)-~2|ρw
      (-t)↓t↓({+/,w×ω}⌺(ρw)⊢I)
```

The variable t is the number of elements that have to be dropped. Per `⌺` semantics, one is subtracted from the even elements of the weight vector's shape, and that result is halved: padding is added at the beginning and end of each axis. The padding is dropped this way: `(-t)↓t↓` — drop t from the end and drop t from the beginning. Finally, `⌺(ρw)⊢I` slides a window, of shape identical to w, over the array I and, at each such position, applies `{+/,w×ω}`, to compute the sum of the element-wise multiplication of w with the corresponding subarray of I.

Dyalog's use of padding introduces significant overhead when w is large, as it spends time computing result elements that must immediately be discarded. Also, as `⌺` is Dyalog-specific and not portable to other APL implementations, we present an alternative formulation:

```
      s ← 1+(ρI)-ρw
      ⊃+/,w×(ιρw){s↑α↓ω}¨⊂I
```

The shape of the resulting array, s, is 1 plus the element-wise difference of the shapes of I and w. We assume that dimensions of I and w are the same. If we are computing a three-element convolution over an n-element vector a with weights $b_0\ b_1\ b_2$, the elements of r would be: $\overset{n-2}{\underset{i=0}{\forall}} r[i] = a[i] \times b_0 + a[i+1] \times b_1 + a[i+2] \times b_2$. Note that i goes over the indices of r, not a. All array languages support element-wise operation on arrays, allowing $\forall i\,.\, c[i] = a[i] + b[i]$ to be written as $a = b + c$, if a, b, c are of the same shape. Hence, we can write $r = a_0 \times b_0 + a_1 \times b_1 + a_2 \times b_2$ where $a_0 = \overset{n-2}{\underset{i=0}{\forall}} a[i]$, $a_1 = \overset{n-2}{\underset{i=0}{\forall}} a[i+1]$, and $a_2 = \overset{n-2}{\underset{i=0}{\forall}} a[i+2]$. This simply means that a_k = `(n-2)↑k↓a`. The indices of b determine how many elements need to be dropped. Generalizing this to an arbitrary number of dimensions produces `(ιρw)`, which generates a nested array of the same shape as w, in

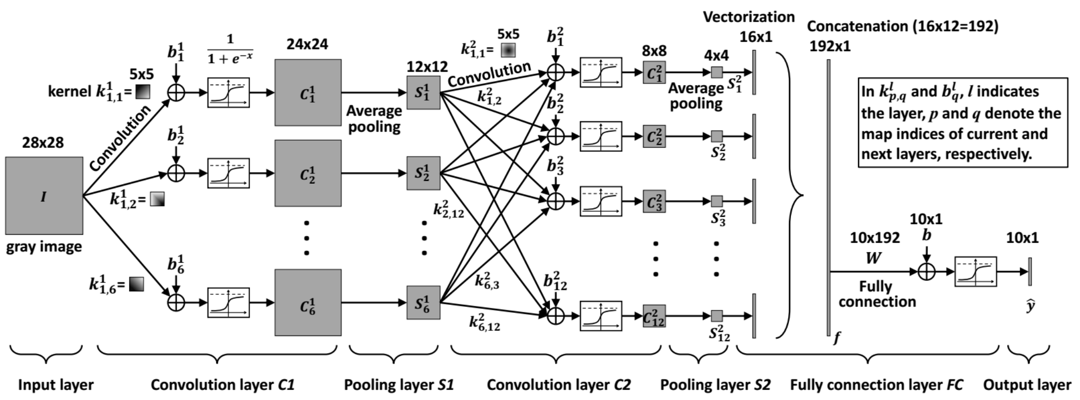

Figure 1. CNN for digit recognition. *The picture is taken from [27]*

which the elements are indices of w, along the lines of what was done in SAC [17] and MoA [12].

For every such index, apply `{s↑α↓ω}`, where the left argument is the index and the right argument is the enclose of I, with the help of the *each* operator, `¨`. This produces an array of shape identical to the shape of w, in which each element is the shifted array I. Each such shifted array is multiplied with the corresponding weight, and that result is summed, then disclosed, to form an element of the final result. When we substitute w and I by formal parameters, and abstract the above expressions as a function, we get:

```
conv←{s←1+(ρω)-ρα⋄⊃+/,α×(ιρα){s↑α↓ω}¨⊂ω}
```

This formulation of convolution is rank and shape polymorphic, as long as the ranks of w and I are the same.

Multiple Convolutions One way to apply `conv` 6 times over I would be to introduce 6 k^1_1 variables, and apply `conv` 6 times. However, a more elegant and generic way is to treat k^1 as a 3d array of shape 6 5 5, in which case we can use the rank operator `⍤`, to apply `conv` to every subarray on n innermost dimensions. For example: `f⍤2⊢k¹` would apply `f` to every 5 5 subarray in k^1. Given that `f` always produces a result of shape s, the shape of `f⍤2⊢k¹` is the catenation of 6 and s.

C_1 prescribes adding a bias to the result of every convolution, so we can use the same pattern: construct b^1 as a 6-element vector, and use the rank operator to add each bias to the corresponding convolution. The semantics of the rank operator allows us to combine these steps, as follows:

```
b¹ {α+ω conv I}⍤(0 2) ⊢k¹
```

For two-argument functions on the left-hand side of the `⍤` we may specify two ranks, one for the left argument and for the right one. In the above example, we add each scalar from `b¹` to every 2d subarray from `k¹`. If we generalize this by not hard-coding the number `2`, which happens to be the rank of I, we obtain a rank- and shape-polymorphic function, which may be usable in other contexts:

```
multiconv←{(I k¹ b¹)←ω⋄b¹{α+ω conv I}⍤(0,(ρρI))⊢k¹}
```

The three arguments `I k¹ b¹` are passed into the function in a nested array, as a right argument.

Activation Functions The last step in C_1 applies the sigmoid (a.k.a standard logistic) activation function to all values. Sigmoid of x is defined as $\frac{1}{1-e^x}$. Since APL defines the unary cases of the arithmetic functions `÷` and `*` as $\frac{1}{x}$ and e^x, respectively (where e is the base of the natural logarithms), we define the activation function as:

```
logistic←{÷1+*-ω}
```

These arithmetic expressions are applicable both to scalars and arrays, so `logistic` can be immediately applied to arrays as well, resulting in a shape-preserving element-wise application of the scalar version of `logistic`.

Average Pooling The average pooling layer S_1 computes the following function for every image C^1_i:

$$\overset{12\ 12}{\underset{u\ \ v=0\ 0}{\forall}} S^1_i[u\ v] = \frac{1}{4}\sum_{m\ \ n=0\ 0}^{2\ 2} C^1_i[2u{+}m\ \ 2v{+}n]$$

That is, split C^1_i into 2×2 non-overlapping blocks, and average the elements in each block. A direct translation of the above formulation into APL would involve array indexing, which is frowned upon in APL dialects, as it is generally inefficient in an interpretive environment. The result shape is the argument shape divided by `2`. For each index in this array, the average of the corresponding elements is:

```
{avg c[(⊂iv) (⊂iv+0 1) (⊂iv+1 0) (⊂iv+1 1)]}¨ι(÷∘2ρc)
```

where `avg` can be defined as the APL train:

```
avg ← {(+/÷≢),ω}
```

which is a shorthand, tacit way to express `(+/,ω)÷(≢,ω)`, in which the left hand side of the division sums all the elements of `ω` in the raveled (flattened) argument array, and the right hand side is the number of elements in the argument array.

One way to simplify the generation of these 4 indices is:

```
{avg c[(⊂2×ω)+ι2 2]}¨ι(÷∘2ρc)
```

This, unfortunately, still generates and uses index vectors, so we shall explore index-free formulations. First, we observe that the split into a grid of 2×2 with further averaging is a stencil operation, where the sliding window moves 2 elements along each axis. This pattern can be expressed with the Dyalog APL stencil operator `⌺`, whereby non-unit movements of the sliding window can be expressed using two-row matrix, in which the first row is the shape of the sliding window, and the second row is the step. In our case, both rows in the matrix are 2 2. According to the padding formula `⌊÷∘2(⍴w)-~2|⍴w`, the stencil of size 2×2 does not create any padding, so we can formulate average pooling as:

```
avgpool ← {÷∘4{+/,⍵}⌺(2 2⍴2)⍤2⊢⍵}
```

That is, for every array in the last two dimensions `⍤2`, apply the stencil operation with sliding window of shape 2×2, moving the window 2 elements along every dimension `⌺(2 2⍴2)`, with the stencil operation that sums all the 4 elements `{+/,⍵}`, dividing that sum by 4. Despite its conciseness, this function uses the Dyalog-specific operator, and it is specific to 4-element average pooling.

We could also do the splitting of a 2d array `c` this way:

```
(x y) ← ⍴c
(x÷2) (y÷2) 2 2 ⍴ ⍉⍤2 ⊢(x÷2) 2 y ⍴ c
```

That is, cut the matrix across the first axis in 2-row chunks `(x÷2) 2 y ⍴ c`. This takes a 2d array of shape x y, producing a 3d array of shape $\frac{x}{2}$ 2 y, then transposes the last two axes: `⍉⍤2`. This gives a result of shape $\frac{x}{2}$ y 2, which can be cut along the second axis by a simple reshape `(x÷2) (y÷2) 2 2 ⍴`. Then, we simply apply `avg` on the last two axes. Finally, this operation itself has to be applied rank-2, as we want to pool the last two dimensions:

```
avgpool ← {
    (x y) ← ⍴⍵
    avg⍤2 ⊢(x÷2)(y÷2)2 2 ⍴ ⍉⍤2 ⊢(x÷2)2y ⍴ ⍵
}⍤2
```

***C_2* and *FC* Layers** Since we represented s as a 3d array of shape $[6, 12, 12]$, the rank-polymorphic specification of `multiconv` is immediately applicable here. Intuitively, if s is a single entity, then all the left hand sides of the arrows from S^1 to C^2 in Fig. 1 merge into a single point, akin to the first convolution. The argument to `conv` is of shape 6 12 12, and each weight $k^2_{i,*}$ is of shape 6 5 5, so the result is of shape 1 8 8. We have 12 $k^2_{i,*}$ and 12 biases, so the result shape of `multiconv` is of shape 12 1 8 8.

A fully connected layer *FC* is just a convolution with a weight identical to the shape of the argument array, so we can use `multiconv` again. Without additional reshapes, the shape of the layer S^2 would be 12 1 4 4. However, as we want to compute 10 weighted sums of all the elements, W becomes of shape 10 12 1 4 4, yielding a result of shape 10 1 1 1 1 from `multiconv`.

Forward Pass At this point, we have enough building blocks to define a "forward path" of the network, *i.e.* Fig. 1, which generates a 10-element vector of probabilities from an image I, with fixed weights k^i and W, and biases b^i and b.

```
forward ← {
  (I k¹ b¹ k² b² W b) ← ⍵
  C¹ ← logistic multiconv I  k¹ b¹
  S¹ ← avgpool C¹
  C² ← logistic multiconv S¹ k² b²
  S² ← avgpool C²
  ŷ  ← logistic multiconv S² W  b
}
```

Each layer is a single multi-dimensional array, and that we compute the *FC* layer directly from S^2. The final answer is the index of $\hat{y}$ with the largest value:

```
{⍵⍳⌈/⍵},ŷ
```

I.e. for the raveled `ŷ`, find the maximum element `⌈/⍵`, and find the index where this element occurs for the first time `⍵⍳`. The `⌈/⍵` produces the two-argument (dyadic) maximum function `⌈` over the array, and the two-argument version of `⍳` with the array on the left and the element on the right returns the first occurrence of this element in that array, which we abstract into this function:

```
maxpos ← {(,⍵)⍳⌈/,⍵}
```

3.1 Gradient Descent

Training of the network entails propagation of the recognition error back into weights and biases, which we do using *stochastic gradient descent.* The overall idea is to consider $\hat{y}$ as a function over weights and biases. The objective function we want to minimise is: $o = \frac{1}{2}\sum_i(\hat{y}_i - y_i)^2$, where y is the correct answer, *i.e.* a 10 element vector with value 1 at the position that corresponds to the digit depicted in I and zeroes at all the other positions. Due to the linear nature of the network, and carefully chosen activation functions, computation of partial derivatives makes a great use of the chain rule. For example, the derivative of our objective function over some weight w would be:

$$\frac{\partial o}{\partial w} = \sum_i \frac{\partial o}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_j}{\partial w} = \sum_i (\hat{y}_i - y_i)\frac{\partial \hat{y}_i}{\partial w}$$

The chain rule can then be applied again, allowing us to "reverse" each layer and reconnect them by inverting the arrows in Fig. 1. For precise mathematical details, please refer to the Section 2 of [27]. In the rest of this section, we focus on the implementation of these back layers.

Mean Squared Error Our objective is the sum of squared differences divided by two. This can written as a direct translation of the above sentence:

```
meansqerr ← {÷∘2+/,(⍺-⍵)*2}
```

Note that we ravel (flatten) `(α-ω)*2` to produce a 10-element vector from ŷ, obtaining a scalar (zero-dimensional array) as a result of sum-reduction.

Back Sigmoid The derivative of the sigmoid function is:

$$\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

Since our sigmoid functions are always "between" the layers, the derivative of sigmoid will be always in a larger derivative chain. Therefore, for convenience, we add a multiplier for that part of the chain:

```
blog ← {α×ω×1-ω}
```

Back Average Pooling If 2d array s is the average pooling of t, then each s_i is defined as $\frac{1}{4}(t_i + t_j + t_k + t_l)$ for some indices j, k, l. Any partial derivative of s_i over t_i is simply $\frac{1}{4}$. Therefore, backpropagation through the average pooling layer can be computed as: $\Delta t[i\ j] = \frac{1}{4}\Delta s[\lfloor i/2 \rfloor\ \lfloor j/2 \rfloor]$. Higher-rank arguments apply this expression on the innermost 2d planes, elegantly expressed in APL as:

```
backavgpool ← {2⌿2/ω÷4}⍤2
```

This reads: for a given argument array `ω`, divide every element by 4, replicate every column twice (`2/`), and then replicate each row twice (`2⌿`). Finally, apply this function rank-2. `backavgpool` hardcodes the size of the pooling, but we can, in principle, provide a left argument of pool size to give a generalized pooling, *e.g.*

```
{(x y)←α ⋄ {x⌿y/ω÷x×y}⍤2 ⊢ω}
```

Back convolution The linear nature of the convolution implies that the derivatives are constants, namely the input of the convolution itself. Hence, we can approximate the error in the weights as a convolution of the input with the error. So, there is no need for a separate function.

The error of the bias is a sum of the error we are propagating from the previous stages of the chain, since the derivative of the bias is constant 1.

```
backbias ← {+/,ω}
```

This function is so trivial that we consider it nugatory, as its inline definition, at three characters, is shorter than its name, and the meaning of the expression is immediately evident. We keep the definition as it represents a logical step in the algorithm.

To compute the propagation of the error into the input of the convolution, note that the derivatives of the convolution with respect to the input are simply weights, with some care taken to describe the index relation between the value we are propagating and the index of the input that it affects. Consider the following example:

```
      in w d
┌───────────┬───┬─────┐
│0.5 1 1.5 2│1 2│1 2 3│
│2.5 3 3.5 4│3 4│4 5 6│
│4.5 5 5.5 6│   │7 8 9│
│6.5 7 7.5 8│   │     │
└───────────┴───┴─────┘
```

This shows three arrays: `in` is the argument. Its values are not relevant, but they are useful in talking about positions; `w` are weights and `d` is the propagated error. The convolution is performed in reverse, to determine which weights contribute to which positions. It is clear that weight `1` contributes to the positions of `in` from `0.5` to `5.5`, as multiplication of any other element with the weight 1 implies out of bound access. By the same reasoning, weight 2 contributes to positions from `1` to `6`; weight 3 from `2.5` till `7.5` and weight 4 from `3` till `8`. This pattern can be expressed using the concept of overtake: a take of more elements than there are in the array pads the result with zeroes:

```
      5 ↑ 1 2 3
1 2 3 0 0
      ¯5 ↑ 1 2 3
0 0 1 2 3
```

This means that we can overtake `d` at every weight index in the following way:

```
      ,w{(⍴in)↑(-ω+⍴d)↑d}¨⍳⍴w
┌───────┬───────┬───────┬───────┐
│1 2 3 0│0 1 2 3│0 0 0 0│0 0 0 0│
│4 5 6 0│0 4 5 6│1 2 3 0│0 1 2 3│
│7 8 9 0│0 7 8 9│4 5 6 0│0 4 5 6│
│0 0 0 0│0 0 0 0│7 8 9 0│0 7 8 9│
└───────┴───────┴───────┴───────┘
```

I.e., pad with zeroes on the top/left with `(-ω+⍴d)↑`, then trim the right/bottom with `(⍴in)↑`, then multiply shifts by the corresponding weight and sum that result:

```
backin ← {(d w in)←ω ⋄ ⊃+/,w{(⍴in)↑(-ω+⍴d)↑α×d}¨⍳⍴w}
```

Finally, the above three steps are combined to create a function that propagates weight-, bias-, and input-errors:

```
bmconv ← {
  (δo ws in bs) ← ω
    d ← ⍴⍴in
  δin ← +⌿δo {backin α ω in} ⍤(d, d) ⊢ ws
  δws ← {ω conv in} ⍤d ⊢ δo
  δbs ← backbias ⍤d ⊢ δo
  δin δws δbs
}
```

As can be seen, this is mainly bookkeeping. We apply `backin` rank-`d`, then sum along the leading axis. This builds on the assumption that convolution evaluates a vector of outputs. If rank increases, the enclose/disclose approach can be used to sum the subarrays. Back-weights and back biases are simply applications of the corresponding functions using rank. Putting all the basic blocks together, we have:

```
train ← {
  (I y k¹ b¹ k² b² W  b) ← ω
           C¹ ← logistic multiconv I  k¹ b¹
```

```
            S¹ ← avgpool C¹
            C² ← logistic multiconv S¹ k² b²
            S² ← avgpool C²
             ŷ ← logistic multiconv S² W  b
            δŷ ← ŷ - y
             e ← ŷ meansqerr y
   (δS² δW δb) ← bmconv (δŷ  blog  y) W  s² b
           δC² ← backavgpool δS²
 (δS¹ δk² δb²) ← bmconv (δC² blog C²) k² S¹ b²
           δC¹ ← backavgpool δS¹
   (_ δk¹ δb¹) ← bmconv (δC¹ blog C¹) k¹ I  b¹
 δk¹ δb¹ δk² δb² δW δb e
}
```

4 Performance

Although our paper focuses on APL's productivity as a specification language, it is enlightening to compare execution times of our APL-based CNN implementation against ones that use state of the art frameworks.

Experimental Setup We conducted timing tests on a 16GB Intel i7-6700HQ CPU, at 2.60GHz running Gentoo Linux, kernel version 5.0.0. We used Dyalog APL versions 16.0.32742 and 17.0.35814, SaC compiler version 1.3.3-MijasCosta-334-geedc7, and TensorFlow version 1.13.0. We present arithmetic means of timing for system initialisation, training 1000 images, updating weights at every image, and recognising 10000 images.

Experimental Result Analysis The elapsed times for our CNN performance measurements are given in Figure 2.

Environment	Init (s)	Train 10^3 (s)	Test 10^4 (s)
APL	0.3	16.3	33.6
TensorFlow	6.0	0.85	6.3
SaC	0.4	0.65	2.2

Figure 2. Elapsed times (seconds) for CNN benchmark

As can be seen, in comparison to TensorFlow, the APL version is about 20 times slower when training and about 5 times slower when recognising the images. APL's initialisation is fast, as it reads four input files, the largest of which is the 60000 training images, occupying about 47MB. TensorFlow, by contrast, takes much longer to initialise, although this rarely presents a problem when running real-world neural networks, because long training times dominate execution time.

The execution time difference is unfortunate, and this may be part of the price for concise framework-less specification. Historically, the highly interactive nature of APL interpreters has limited the scope of performance improvements, as syntax classes can change underfoot during execution: *e.g.* verbs can become nouns, or vice versa. Secondly, generation of large number of nested arrays drags APL performance down by orders of magnitude, due to the load placed on the array memory subsystem. Finally, APL interpreters typically do not perform simple optimizations, such as code motion to lift loop-invariant expressions, because of a chosen interpreter design decision to allow interruption and continuation of evaluation, an interactive session feature that few other systems support. The absence of special treatment of the convolution operator and lack of just in-time compilation also contribute to poor performance.

At this stage, we could say that APL is nothing but a prototyping language, but the SaC [18] runtime brings us hope. The SaC runtime reflects the performance of our hand-translated version of the APL code into SaC, a functional array-oriented language for high-performance. It turns out that a very similar specification can bridge the performance gap between APL and TensorFlow, with APL potentially even outperforming the latter. All that remains to do is to automate translation of APL into a compiled language like SaC.

Our initial attempts to use the APEX compiler [3] were not successful, as APEX did not support dfns and nested arrays. The current version of APEX includes a simple Dfn-to-TradFn converter, as well as support for stranded function arguments and results. Work to replace remaining uses of nested arrays in the CNN code is underway as of this writing, opening up immediate perspectives for future work. As an added bonus, when generating SaC code, we can immediately obtain parallel execution on multi-core architectures, as well as GPUs, by simply providing a flag to a compiler.

5 Related Work

In this section, we briefly overview state of the art machine learning networks and the landscape of array languages.

Machine Learning Frameworks State of the art machine learning frameworks, such as TensorFlow [1], Caffe [10], CNTK [25], Torch [5], and PyTorch [15] have similar overall designs: a core part of the framework is written in C/C++, and an interface part is written in a dynamic scripting language, like Lua or Python. The interface wraps around the core components to provide a convenient glue for connecting highly-optimised machine-learning primitives for linear algebra and tensor operations.

The core components are pre-optimised for the range of architectures including multi-core CPUS, GPUs, and distributed systems. TensorFlow also supports custom hardware known as Tensor Processing Units.

All the frameworks attempt to optimise the algorithm prior to running it. In these frameworks, the neural network has an internal representation in the form of a dataflow graph, in which the graph nodes are computation layers, and its edges are input tensors for the given layer. Optimisation on the dataflow graph makes it possible to merge sequences of operations into larger and potentially more efficient operations, and to run concurrent nodes in parallel. The availability of the dataflow graph gives rise to automatic differentiation, which simplifies expressibility of the specification.

Array Languages A number of array language, including J [20], K [24], and Nial [11], follow APL's design. They treat every object as an array (maybe except functions) and provide a large subset of the APL operators. Typically, these languages are interpreted, which limits potential optimisations and performance, yet maintaining excellent interactivity.

The untyped nature of APL implies that nearly all errors are detected only at runtime. In contrast, SaC, Futhark [7], Remora [19] and Qube [22] are array-oriented functional languages with static type systems, which makes it possible to detect range and domain errors prior to program execution. Instead of providing built-in APL operators natively, these languages offer low-level constructs which are typically sufficient to implement APL operators as library functions. The compilers provided for these languages focus on generating high-performance code for various architectures, including multi-core CPUs and GPUs.

Finally, Matlab [21], Julia [4] and Python [23], with its Numpy [13] library, offer some notion of multi-dimensional arrays and a subset of APL operators. These are embedded in the context of a general-purpose imperative language. All the mentioned languages come with interpreters only and rarely exhibit exceptional levels of performance, yet, like APL, they are very useful for quick prototyping.

6 Conclusions & Future Work

In this paper we demonstrated how CNNs can be implemented in APL, resulting in a concise framework-free specification that is immediately executable. The CNN-specific building blocks can be implemented in 10 lines of APL. The resulting primitives are rank- and shape-polymorphic, making them immediately applicable in other contexts.

Once the building blocks are implemented, domain experts are in a similar situation as when they use machine-learning frameworks, such as TensorFlow. The primitives look similar to the TensorFlow ones, and they have to be combined into networks that can be trained and executed. One noticeable difference though is the availability of automatic differentiation in TensorFlow, which noticeably simplifies specification of back propagation.

Our proposed APL approach exposes the precise implementation of the framework, and due to its concise nature, enables domain experts to examine and alter the framework. In contrast, finding out actual implementation details of frameworks such as TensorFlow is challenging, despite source code being available. The main reasons for this are: a large volume of code, complex, multi-layered design, and dependencies on a number of external libraries. Minor extensions are typically allowed by the framework design, whereas substituting core parts, or changing low-level details like optimisations or computational behaviour is typically not feasible for people that are not deeply involved in the implementation of the framework under consideration.

The key to the APL's conciseness is the combinatorial nature of the language. None of our operators use explicit indexing; this improves clarity and reduces the potential for mistakes in the specification. Also, as all our combinators are purely functional, the resulting specification can be easily ported to non-APL contexts.

This sheds some light on using APL as a framework for shallowly-embedded machine learning DSLs. We can clearly see that this is possible; however, our runtime experiments suggest that existing tools fail to deliver the levels of performance comparable with the state of the art frameworks. As we have seen in our experiments, this is a technical problem, rather than a conceptual one, and one that can be solved by using compilation technologies. This brings us to future work.

As noted, we are addressing the absence of dfn and nested array support in the APEX compiler, and intend to verify whether compiler-generated SaC code will perform as well as the hand-written version that we used for measurements. At the same time, we are going to investigate the code generated by the SaC multi-thread and GPU backends, to ensure that they deliver significant speed-ups on those architectures. We also expect that replacing nested arrays usage in the APL code by uses of the rank conjunction will simplify the task of compiling that code, and will certainly make it perform considerably faster in an interpreted environment.

Finally, we would like to investigate cleaner ways for users to extend APL interpreters — ones that do not require the assistance of an APL implementer. For example, a pseudo-primitive, perhaps written in C or SaC, to implement the idiom `s↑α↓ω`, would reduce the number and size of intermediate arrays used in `conv`. This may be possible to do via the `⎕NA` interface, provided by Dyalog APL.

Similarly, potential performance benefits from invoking Dyalog APL's JIT facility are yet to be investigated.

Acknowledgments

We offer our thanks to the program committee and anonymous reviewers for their thoughtful insights and constructive comments on our paper. This work was supported in part by grants EP/N028201/1 and EP/L00058X/1 from the Engineering and Physical Sciences Research Council (EPSRC)

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga,

Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Robert Bernecky. 1987. An Introduction to Function Rank. *ACM SIGAPL Quote Quad* 18, 2 (Dec. 1987), 39–43.

[3] Robert Bernecky. 1997. *APEX: The APL Parallel Executor.* Master's thesis. University of Toronto.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[5] Ronan Collobert, Koray Kavukcuoglu, Clément Farabet, et al. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, Vol. 5. Granada, 10.

[6] Dyalog Limited 2019. *Dyalog APL Language Reference Guide* (Dyalog version 17.0 ed.). Dyalog Limited, Bramley, Hampshire, UK. http://docs.dyalog.com/17.0/Dyalog%20APL%20Language%20Reference%20Guide.pdf

[7] 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.* ACM, 556–571.

[8] Sakshi Indolia, Anil Kumar Goswami, S.P. Mishra, and Pooja Asopa. 2018. Conceptual Understanding of Convolutional Neural Network- A Deep Learning Approach. *Procedia Computer Science* 132 (2018), 679 – 688. https://doi.org/10.1016/j.procs.2018.05.069 International Conference on Computational Intelligence and Data Science.

[9] Kenneth E. Iverson. 1979. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (Aug. 1979).

[10] 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. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[11] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. 1984. Nial: A Candidate Language for Fifth Generation Computer Systems. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge (ACM '84)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/800171.809618

[12] Lenore M. Restifo Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation. Syracuse University.

[13] Travis Oliphant. 2006. NumPy: A guide to NumPy. http://www.numpy.org. [Accessed 2019/02].

[14] Rasmus Berg Palm. 2012. *Prediction as a candidate for learning deep hierarchical models of data.* Master's thesis. Technical University of Denmark.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[16] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85 – 117. https://doi.org/10.1016/j.neunet.2014.09.003

[17] Sven-Bodo Scholz. 1996. *Single Assignment C.* Ph.D. Dissertation. Christian-Albrechts-Universität zu Kiel.

[18] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[19] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

[20] Roger Stokes. 15 June 2015. Learning J. An Introduction to the J Programming Language. http://www.jsoftware.com/help/learning/contents.htm. [Accessed 2019/02].

[21] The Mathworks, Inc., Natick, MA. 1992. *MATLAB Reference Guide.*

[22] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643 – 664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[23] G. van Rossum. 1995. *Python tutorial.* Technical Report CS-R9526. Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[24] Arthur Whitney. 2001. K. http://archive.vector.org.uk/art10010830.

[25] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. 2014. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112* (2014).

[26] Tong Zhang. 2004. Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms. In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML '04)*. ACM, New York, NY, USA, 116–. https://doi.org/10.1145/1015330.1015332

[27] Zhifei Zhang. 2016. *Derivation of Backpropagation in Convolutional Neural Network (CNN).* Technical Report. University of Tennessee, Knoxvill, TN. http://web.eecs.utk.edu/~zzhang61/docs/reports/2016.10%20-%20Derivation%20of%20Backpropagation%20in%20Convolutional%20Neural%20Network%20(CNN).pdf