# Machine Learning in Chapel

Iain Moncrief

August 2023

## 1 Introduction

The Chapel programming language offers many built in constructs for parallel computation. Over the course of the summer, my internship with the Chapel Team focused on implementing and translating machine learning programs from Python to Chapel. An expected result of this process was to asses the difficulty and sufficiency of Chapel for implementing these programs.

## 2 Required Constructs

### 2.1 Tensors

Tensors are the most important data type when it comes to machine learning. They are used to represent any form of agregated numerical data. Given a data structure for tensors, and a sufficient library of operations defined on them, it is possible to program just about any kind of feed forward neural network.

Chapel's generalization of index sets on arrays gives it the status of having tensors built directly into the language. While this specific feature gives Chapel the ability to produce highly performant programs, it sets it apart from other languages. A consequence of Chapel's array index set generalization is that a programmer must consider an array's domain every time they consider the array. While this isn't necessarily a fault of Chapel, it does make the translation of ML programs from Python much less mechanical.

To avoid dealing with domains, I created a Tensor record

```
record Tensor {
    param rank: int;
    type eltType = real;
    var _dom: domain(1,int);
    var data: [this.domain] eltType;
}
```

which wraps an array and only exposes the `rank` and `eltType` type information. This would allow me to define procedures on tensors, without having to consider their domains. For example,

```
proc addMat(a: [?d1] real, b: [?d2] real): [d1] real) where d1.rank == 2 && d2.rank == 2 {
    assert(a.shape == b.shape);
    return a + b;
}
```

would be written as

```
proc addMat(a: Tensor(2), b: Tensor(2)) {
    assert(a.shape == b.shape);
    return new Tensor(a.data + b.data);
}
```

With this interface, it is easy to define methods on tensors that resemble the style of most Python libraries,

```
proc Tensor.magnitude() do
    return sqrt(+ reduce forall x in data do x * x);
proc Tensor.normalize() do
    return this / this.magnitude();
proc Tensor.fmap(f) do
    return new Tensor(f(this.data));
```

### 2.2 Operator Overloading

Having overloaded infix operations on tensors is as much of a necessity as having tensors. This can be done easily in Chapel,

```
operator -(lhs: Tensor(?rank), rhs: Tensor(rank)): Tensor(rank) {
    assert(lhs.shape == rhs.shape);
    return new Tensor(lhs.data - rhs.data);
}
operator -(lhs: Tensor(?rank), c: real): Tensor(rank) do
    return new Tensor(lhs.data - c);
```

Only having operations on primitive data types, like in Java and JavaScript, would make it difficult to express machine learning algorithms.

### 2.3 Objects

Modern ML libraries make use of OOP for representing neural networks. This works by representing each kind of layer as a class that inherits from some generalized layer type.

Chapel has classes and records, which are sufficient for implementing this sort of representation. Though, someone that is coming from Python should note that they must include `shared` before all of their class types, if they expect the behavior that they are used to.

Today, Chapel lacks the ability to store references inside of classes and records. This prevented me from implementing optimizers.

## 3 From Python

Overall, the conversion from Python to Chapel wasn't that difficult. The difficulty came when the Python program relied on language features that Chapel

does not have.

## 3.1 Tensor Library

When it came to translating operations on tensors, such as

```
h,w,c = image.shape
out,kh,kw,in_c = filters.shape # in_c = c
filtered_image = np.zeros(out,*convolve_shape(input_size=(h,w),kernel_size=(kh,kw)))
for i in range(out):
    for j in range(c):
        channel = image[:,:,j]
        kernel = filters[i,:,:,j]
        filtered_image[i,:,:] += convolve(channel, kernel)
```

the corresponding Chapel code is not much different,

```
const (h,w,c) = image.shape;
const (outC,kh,kw,inC) = filters.shape;
const newShape = convolveShape((h,w),(kh,kw));
var filteredImage: Tensor(4) = tn.zeros(outC,(...newShape),inC);
forall (i,j) in {0..#outC,0..#inC} with (+ reduce filteredImage) {
    const channel: Tensor(2) = image[..,..,j];
    const kernel: Tensor(2) = filters[i,..,..,j];
    filteredImage[i,..,..] += convolve(channel,kernel);
}
```

thanks to Chapel having a rich syntax.

As you might expect, the ease comes from having an implementation of the functions you need. Chapel has a built in linear algebra module that implements many useful operations on arrays. Though, I found it easier to just re-implement them for the `Tensor` record, since the conventions that it uses are inconsistent with the rest of Chapel and the documentation is vague. If Chapel was to devote some time to replicating the methods that NumPy and PyTorch have, programming neural networks would be significantly easier. From all of the articles and ML programs that I have read through, it seems like ML people *really* don't want to touch the implementation of these operations. So this effort would be good for both sides, since the ML people would have a fruitful tensor library that would make Chapel accessible to them, and the Chapel team (who know this best) could provide the most efficient implementations, helping Chapel become the most attractive language for writing performant ML programs.

## 3.2 Automatic Differentiation System

PyTorch and TensorFlow both implement an automatic differentiation system for tensors. This is one of the most attractive aspect of these libraries. It allows users to operate on tensors and *automatically* obtain the partial derivatives of the results with respect to any other tensor used in the computation.

Specifically, suppose $X_1, \ldots, X_n, Y \in \mathbb{T}$ are tensors, where $\mathbb{T} = \bigcup_{k=1}^{\infty} \mathbb{R}^k$, and $f : \mathbb{T}^n \to \mathbb{T}$ is an $n$-ary operation on tensors. Then if

$$Y \leftarrow f(X_1, \ldots, X_n)$$

there would be a method on $Y$ named `backward()`, which would populate all $X_1, \ldots, X_n$ with a field `grad`, via

$$X_i\texttt{.grad} \leftarrow \frac{\partial Y}{\partial X_i} \qquad 1 \leqslant i \leqslant n.$$

This system allows users to define any sort of computation on tensors and obtain the derivative of that computation for free (provided they used the operations defined in PyTorch or TensorFlow). I tried to replicate this system in Chapel, but it relies on closures and object references inside of records. Having an automatic differentiation system for tensors implemented in Chapel would make writing ML programs exponentially easier. Note for Daniel, it is really $\mathbb{T} = \bigcup_{S \in F(\mathbb{N})} \times_{i=1}^{|S|} \mathbb{R}^{S_i}$.

*It looks like the following already exists, but it could be developed more:* Having a well maintained system for calling Chapel code via Python would be incredibly impactful. It would allow people to use Chapel to the degree that they please so that they can *ease in* to the language and not leave behind all of the libraries that are familiar to them. I think this would be very attractive to the Python community since Python is strictly single threaded and known to be slow. And since it is so easy to write parallel programs in Chapel, Python programmers would be compelled to try it.

### 3.3 Alternative to Tensor Record

As I was going through Jeremiah's and Engin's comments on my pull request (https://github.com/chapel-lang/chapel/pull/23140), I realized that there were significant performance drawbacks to having tensors be represented by a record that holds an array. Since all of the operations I have implemented can be instead defined on the internal `_array` type, it *might* be better to have the `Tensor` module just add operations to arrays. From here, the only benefit that `Tensor` provides is the ability to specify the rank in the type signature.

I propose that Chapel adds a syntax to support this for their built in arrays, such as,

```
proc f(t: [1] real): [2] real { ... }
// or
proc f(t: [{1}] real): [{2}] real { ... }
// or
proc f(t: [domain(1)] real): [domain(2)] real { ... }
// or
proc f(t: tensor(1,real)): tensor(2,real) { ... }
```

to mean

```
proc f(t: [?dIn] real): [domain(2)] real where dIn.rank == 1 { ... }
```

so that if users want to restrict the ranks of the domains (e.g. matrix-vector multiplication), they don't have to string param expressions together via `&&` in the **`where`** clause. This would make it much neater to program with tensors with the built in Chapel array representation. This is what I would like to see most in the next release.

### 3.4 Overloading

An aspect of Chapel that I think Python programmers will appreciate is its type system. Chapel has a powerful static type system that emphasizes generiticity. Its approach uses types to distinguish the specificity of different method implementations. While Python hasn't evolved in the presence of types, this concept is very familiar. Consider the Python function

```
def my_len(x):
    if isinstance(x,list):
        return len(x)
    elif isinstance(x,dict):
        return len(x.keys())
```

which differentiates the implementation of `my_len` based on its type. In Chapel, this is done by providing two different definitions with differing types. The analogous Chapel code would be

```
proc myLength(x: list) do
    return length(x);
proc myLength(x: map) do
    return length(x.keys);
```

### 3.5 Plotting Library

Matplotlib makes it very easy to visualize data in Python. Most ML projects will use Matplotlib in some form. I think that Chapel should have some kind of data visualization module, so that users don't have to implement a gnuplot wrapper.

### 3.6 Additional Binary Operators

This would be a nice-to-have: additional operators that users can be free to overload. In Python, matrix multiplication can be written as `A @ B` while retaining `A * B` for element-wise multiplication. This could be an opportunity for Chapel to consider adding some cool operators on domains as described in https://homes.cs.washington.edu/ bradc/cv/pubs/conferences/apl99.pdf . But again, not that big of a deal.