Extracted from:

Machine Learning in Elixir

Learning to Learn with Nx and Axon

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Machine Learning in Elixir

Learning to Learn with Nx and Axon

Your Elixir Source

Sean Moriarity

Edited by Tammy Coron

Machine Learning in Elixir

Learning to Learn with Nx and Axon

Sean Moriarity

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 979-8-88865-034-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—July 19, 2023

Understanding Nx Tensors

Start by running the following code in a new Livebook cell:

```
Nx.tensor([1, 2, 3])
```

Which generates the following output:

```
#Nx.Tensor<
  s64[3]
  [1, 2, 3]
>
```

You've just created a tensor using one of Nx's creation methods. Nx.tensor/2 is the easiest way to create a tensor from a number, a list of numbers, or a nested list of numbers. Try creating a few more tensors using the following code:

```
a = Nx.tensor([[1, 2, 3], [4, 5, 6]])
b = Nx.tensor(1.0)
c = Nx.tensor([[[[[[1.0, 2]]]]]])
IO.inspect a, label: :a
IO.inspect b, label: :b
IO.inspect c, label: :c
```

Which returns:

```
a: #Nx.Tensor<
  s64[2][3]
  [
    [1, 2, 3],
    [4, 5, 6]
  ]
>
b: #Nx.Tensor<
  f32
  1.0
>
c: #Nx.Tensor<
  f32[1][1][1][1][1][2]
  [
    [
      [
        [
          [
            [1.0, 2.0]
          ]
        ]
      ]
    ]
  ]
```

```
>
```

You'll see three properties of a tensor every time you inspect its contents: the tensor's type, the tensor's shape, and the tensor's data. All of these properties make a tensor distinctly different from a generic Elixir list.

Tensors Have a Type

Create and inspect two tensors by running the following code in a Livebook cell:

```
a = Nx.tensor([1, 2, 3])
b = Nx.tensor([1.0, 2.0, 3.0])
IO.inspect a, label: :a
IO.inspect b, label: :b
```

You'll see the following output:

```
a: #Nx.Tensor<
  s64[3]
  [1, 2, 3]
>
b: #Nx.Tensor<
  f32[3]
  [1.0, 2.0, 3.0]
>
```

Do you notice a difference between the two tensors, aside from the difference in data?

Notice that tensor `a` displays `s64`, whereas tensor `b` displays `f32`. Both `s64` and `f32` are the *numeric type* of the tensor's data. If you've worked with types in programming languages before, you're likely familiar with some of the numeric types Nx offers.

Nx types dictate how the underlying tensor data is interpreted during execution and inspection. You'll see in Tensors Have Data, on page 9 that tensor data is not represented in an Elixir list, but instead as raw bytes. The tensor's type tells Nx how to interpret those raw bytes.

Tensor types are defined by a type class and a bit width. The type class can be a signed integer, unsigned integer, float, or brain float. Signed and unsigned integers can have a bit-width of 8, 16, 32, or 64. Floats can have bit widths of 16, 32, or 64. Brain floats can only have a bit width of 16. Brain floats are a special type of floating point number optimized for deep learning. You can specify types when creating tensors using a tuple of `{class, bit-width}`. The following table illustrates each type, their Elixir representation, and their inspected string representation:

Class	Widths	Elixir Representation	String Representation
signed integer	8, 16, 32, 64	{:s, 8}, {:s, 16}, {:s, 32}, {:s, 64}	s8, s16, s32, s64
unsigned integer	8, 16, 32, 64	{:u, 8}, {:u, 16}, {:u, 32}, {:u, 64}	u8, u16, u32, u64
float	16, 32, 64	{:f, 16}, {:f, 32}, {:f, 64}	f16, f32, f64
brain float	16	{:bf, 16}	bf16
complex	64, 128	{:c, 64}, {:c, 128}	c64, c128

Notice that you specify a type's class with an *atom*. If you're familiar with Elixir, you know an atom is a constant whose values are their own name. You then specify the type's bit width. A type's bit-width dictates the size each element of the tensor occupies in memory. For example, each element in a signed 64-bit tensor occupies 64-bits or 8 bytes of memory. Larger bit-width types are more *numerically precise*, which means you can represent a larger range of values and not worry as much about *underflow* or *overflow*. Underflow occurs when you try to represent a value that is too small for a computer to represent in storage. For example, create a tensor with the following code:

```
Nx.tensor(0.000000000000000000000000000000000000000000001)
```

You'll see the following output:

```
#Nx.Tensor<
  f32
  0.0
>
```

But you didn't want to create a tensor with a value of `0.0`; you wanted a tensor with a value of `1.0e-45`. The value you're trying to represent is underflowed to `0.0`. A 32-bit float is not precise enough to store your value. If you increase the bit width to 64, you'll be able to properly represent the number you want. You can tell Nx to use a specific type by passing the `:type` option to `Nx.tensor/2`:

```
Nx.tensor(1.0e-45, type: {:f, 64})
```

Running the code above will return:

```
#Nx.Tensor<
  f64
  1.0e-45
>
```

A 64-bit float occupies more memory, and is thus able to store a larger range of numbers. Overflow occurs when the number you are trying to store is too large for the given type. This happens often with low precision integer types. For example, create a tensor using the following code:

```
Nx.tensor(128, type: {:s, 8})
```

Here, you're trying to create a tensor with a value of 128 and a type of `{:s, 8}` or a signed 8-bit integer tensor. After running the code, you'll see:

```
#Nx.Tensor<
  s8
  -128
>
```

That's surprising! Signed 8-bit occupy 1 byte of memory and can only represent values between `-128` and `127`. Anything outside of that range will be squeezed to some value within the supported range, which results in the behavior you see here.

Precision issues are very common in machine learning because you're often working with floating-point types. Floating-point types attempt to capture a large range of real values. However, it's not possible to fit an infinite range of numbers into a finite amount of storage. You will sometimes see surprising due to precision issues. Throughout this book, you'll see code examples that attempt to work around the limitations of floating-point numbers.

As you may have noticed, tensors have a homogenous type. For every tensor you've created, there's always been a single type. You cannot have tensors with mixed types. Nx will choose a default type capable of representing the values you are trying to use when you create a tensor, unless you explicitly state otherwise by passing a `:type` parameter. You can see this default typing in action by running the following code:

```
Nx.tensor([1.0, 2, 3])
```

Which returns:

```
#Nx.Tensor<
  f32[3]
  [1.0, 2.0, 3.0]
>
```

Even though the last two values are integers, Nx cast them to floats because the highest type was a floating-point value, and Nx did not want you to unnecessarily lose precision.

Having homogenous types in an array programming library like Nx is necessary for a couple of reasons. First, having homogenous types eliminates the need to store additional information about every value in the tensor. Second, having homogenous types enables unique optimizations for certain algorithms. For example, imagine you want to compute the index of the maximum value in a tensor of type `{:s, 8}`. Because you know that every value in the tensor is a signed 8-bit integer, you also know that the maximum possible value is 127. If you ever observe 127 in the tensor, you can halt the algorithm without traversing the rest of the tensor because 127 is *guaranteed* to be maximal. Type-specific optimizations, such as this one, are very common in numerical computing.

Tensors Have Shape

You've probably noticed the nested list representation of data when inspecting the contents of a tensor. However, tensor data is not stored as a list at all. The nesting you see during inspection is actually a manifestation of the tensor's *shape*. A tensor's shape is the size of each dimension in the tensor. Consider the following tensors:

```
a = Nx.tensor([1, 2])
b = Nx.tensor([[1, 2], [3, 4]])
c = Nx.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

If you inspect each tensor with the following code:

```
IO.inspect a, label: :a
IO.inspect b, label: :b
IO.inspect c, label: :c
```

You'll see the following output:

```
a: #Nx.Tensor<
  s64[2]
  [1, 2]
>
b: #Nx.Tensor<
  s64[2][2]
  [
    [1, 2],
    [3, 4]
  ]
>
c: #Nx.Tensor<
  s64[2][2][2]
  [
    [
      [1, 2],
      [3, 4]
    ],
    [
      [5, 6],
```

```
      [7, 8]
    ]
  ]
>
```

Notice the value next to each tensor's type. That value is its shape. Shapes in Nx are expressed using tuples of positive integer values. The representation you see in the previous code example is just a pretty-printed version of each tensor's shape. Tensor `a` has a shape of `{2}` because it has one dimension of size 2. Tensor `b` has shape `{2, 2}` because it has two dimensions, each of size 2. Finally, tensor `c` has a shape of `{2, 2, 2}` because it has three dimensions, each of size 2. Notice as the number of dimensions increases, so does the level of nesting in the inspected data.

The number of dimensions is typically referred to as the tensor's *rank*. Again, if you're coming from a mathematical background, this use of the word rank might confuse you. In the world of numerical computing, the rank just corresponds to the number of dimensions or the level of nesting in the tensor. *Scalars* do not have any level of nesting at all because they don't have any shape. You can think of a scalar as just a 0-dimensional tensor. A scalar is a single value. Run the following in a new cell:

```
Nx.tensor(10)
```

And you'll see the following output:

```
#Nx.Tensor<
  s64
  10
>
```

Notice there is no output where the shape typically is shown. That's because this is a scalar tensor, and therefore, it has no shape.

So, why do tensors need to have a shape? Remember, the point of tensors is to have a flexible numeric representation of the outside world. If you were to try to represent an image with no semblance of shape, it would be very difficult.

Imagine you have a 28x28 RGB image. Images are typically represented with a shape `{num_images, height, width, channels}` where `channels` corresponds to the number of color channels in the image - 3 in this case for red, green, and blue color values. If you were asked to access the green value of the 10th pixel down and the 3rd pixel towards the center of the image, how would you do that, given only a flat representation of the image? It wouldn't be possible. You would have no idea how the image is laid out in memory. Without any information as to the shape of the image, you can't even be sure how many color channels the image has or what the height and width of the image are.

A tensor's shape helps you naturally map tensors to and from the real world. Furthermore, a tensor's shape tells you how to perform certain operations on the tensor. For example, if tensors did not have any shape, there would be no way to perform matrix multiplications between two tensors because you would have no understanding of the size of each dimension in your matrices.

To more naturally map a tensor's shape to the real-world, Nx implements the concept of *named tensors*. Named tensors introduce dimension or axis names for more idiomatic tensor manipulation. For example, if you have an image, you might have dimension names of :height, width, and :channels. Each dimension name is an atom. You can use dimension names to perform operations on specific dimensions. You can specify the names of a tensor on creation. For example, running the following code:

```
Nx.tensor([[1, 2, 3], [4, 5, 6]], names: [:x, :y])
```

Will return the following output:

```
#Nx.Tensor<
  s64[x: 2][y: 3]
  [
    [1, 2, 3],
    [4, 5, 6]
  ]
>
```

Notice the shape representation now tells you the size and name of each dimension. Rather than saying dimension 1, you can say dimension :y. Named dimensions give semantic meaning to otherwise meaningless dimension indices.

Tensors Have Data

As previously mentioned, tensor data is stored as a byte array or an Elixir *binary*. A binary is an array of character bytes. These bytes are interpreted as a nested list of values depending on the tensor's shape and type. Representing tensor data in this way helps simplify many Nx implementations. When you create a new tensor using Nx.tensor/2, Nx traverses the values in each list and rewrites the value in a binary representation. To view this binary representation, create a tensor with the following code:

```
a = Nx.tensor([[1, 2, 3], [4, 5, 6]])
```

Now, get the underlying binary representation using Nx.to_binary/1:

```
Nx.to_binary(a)
```

The results of which will be:

```
<<1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3,
  0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 5,
  0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0>>
```

Notice the binary representation has no semblance of shape or type. It's literally just a flat collection of byte values. Because Nx has to turn your data into a binary representation when you use Nx.tensor/2, it's more performant to, instead, create tensors using Nx.from_binary/2:

```
<<1::64-signed-native, 2::64-signed-native, 3::64-signed-native>>
|> Nx.from_binary({:s, 64})
```

The <<>> syntax creates an Elixir binary. Note you can construct binaries in the style shown using binary modifiers. The code above creates the following tensor:

```
#Nx.Tensor<
  s64[3]
  [1, 2, 3]
>
```

Nx.from_binary/2 takes a binary and a type and creates a 1-dimensional tensor from the binary data. You can change the shape of the tensor using Nx.reshape/2:

```
<<1::64-signed-native, 2::64-signed-native, 3::64-signed-native>>
|> Nx.from_binary({:s, 64})
|> Nx.reshape({1, 3})
```

Which returns:

```
#Nx.Tensor<
  s64[1][3]
  [
    [1, 2, 3]
  ]
>
```

Notice the usage of the native binary modifier. Because Nx operates at the byte level, *endianness* matters. Endianness is the order in which bytes are interpreted or read in the computer. The native modifier tells the VM to use your system's native endianness. If you're attempting to read binary data from a computer with different endianness than your machine's, you might run into some problems. For the most part, you shouldn't have to worry, but it's something to be conscious of if you need to work with a tensor's raw data.

Tensors Are Immutable

One notable distinction between Nx and other numerical computing libraries is that Nx tensors are immutable, which means that none of Nx's operations change the tensor's underlying properties. Every operation returns a new tensor with new data every time. In some situations, this can be expensive. However, Nx overcomes the limitation of immutability by introducing a programming model that enables Nx operator fusion. You'll use this programming model in Going from def to defn, on page ?.