grox.io
Learning

Elixir NX

The Machine Learning Toolbox

Bruce A. Tate
Edited by Jacquelyn Carter

ELIXIR NX

Bruce A Tate

Table of Contents

Elixir Nx: The Machine Learning Toolbox ... 1
1. This is a Beta Book ... 2
    1.1. Beta 1.0: September 1, 2021 ... 2
    1.2. Provide Feedback ... 2
2. Nx Means Tensors ... 3
    2.1. Document, Share, and Explore With Livebook ... 5
    2.2. Plot Elixir Data in a Livebook ... 9
    2.3. Nx and Tensors ... 12
    2.4. Call Tensor Operators and Functions ... 17
    2.5. What You Built ... 25

Elixir Nx: The Machine Learning Toolbox

© 2021 Bruce A Tate. All rights reserved. Version 1.0.

Published by Grox.io, creator of grox.io.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recoding, scanning or otherwise except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

Editor: Jacquelyn Carter

Cover: Brennan Saucier

ISBN: TBD

1. This is a Beta Book

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

1.1. Beta 1.0: September 1, 2021

Your Numerical Computing Toolbox. Our long-awaited dive into Elixir's numerical computing gets started with a test drive of Nx, and a few of the tools that connect to it. We'll explore notebooks, tensors, and the various mathematical features for dealing with them. In later chapters, we'll use those tools to do some hacking in Axon.

1.2. Provide Feedback

We always welcome feedback from our readers. Let us know what you think or if something is out of place.

To send us feedback, e-mail us at support@grox.io or join our Discord community at https://discord.gg/gxXcZS3c.

2. Nx Means Tensors

Since it’s creation, the Elixir language has been known as a versatile functional language with the ability to succeed in many different environments. The Phoenix project makes it ideal for building scalable and interactive web projects. The Erlang OTP roots give it self-healing properties that make Elixir ideal for backend processing. In short, Elixir is a great language for *showing* and *presenting* data.

However, there’s one slice of the programming landscape that has been off limits to Elixir enthusiasts until recently: numeric computing, or Science, Technology, Engineering, and math (STEM) problems. These STEM problems are grabbing an ever-expanding slice of the total programming profession today. Numerical libraries in Python fuel the machine learning research behind image recognition engines powering smart cars and voice recognition engines. STEM languages like Julia and R help scientists map the universe, point pharmacists to perfect dosages, and give stock brokerages the tools to predict what will happen in the stock market. User interfaces are changing to accept imprecise vocal and handwriting input. Smartphones help users categorize photos by recognizing who is in them. All of these solutions rely on numerical computing.

Elixir’s data structures and numerical limitations made Elixir a poor fit for so-called number crunching, a problem domain characterized by large, mutable arrays of data with specific efficient types. Elixir’s linked lists, immutable data, and numerical representations work well for functional algorithms, but are just about the anthesis of what’s used in typical scientific languages.

In 2020, all of that changed. Sean Moriarity and José Valim announced the Nx project. Rather than creating a new system enabling massive arrays, the team worked on a much more attainable extension. They built an API for typed, multi-dimensional data structures called tensors. Rather than reworking or suplimenting Elixir’s own data structures, they wrapped existing libraries in other languages to access tensors and the numbers that go into them. This wrapping strategy allowed rapid progress.

The Nx project is fundamentally a framework for working with tensors efficiently. The project itself is small. It has a couple of hundred modules, some clever compiler support, and roughs out some core mathematical features that are table stakes in this numerical game.

While the project's footprint is tiny, the overall impact of Nx has been nothing short of earth shattering.

As Nx grows, it's feeding the parallel growth of other Elixir projects long deemed critical for numeric computing. These projects are primarily centered around explorative documentation, generic numerical computing, compilation support for high-performance special-purpose processors, and machine learning. This list will doubtlessly grow rapidly. Many of these projects are growing as a set of sub-projects under an umbrella project called Elixir-Nx. They are:

- Nx. This project provides tensors, compiler support, and specialized functions called `defns` to support the tensors.
- Explorer. A popular Python API for working with tabular data.
- SciData. Example numeric datasets loaded and formatted for Elixir.
- Axon. Machine learning in Elixir.

In addition, the Nx library supports backends for Python's PyTorch and Google's accelerated linear algebra (XLA.) The EXLA project integrates Nx with compiler tools supporting just-in-time compilation and special-purpose processors known as TPUs. Compiler support in Nx can speed the execution of numerical code by orders of magnitude in some cases.

Another project emerged briefly under the Elixir-NX umbrella, but has since become a full Github project in it's own right, Livebook. These notebooks combine data, prose, and code. Coding notebooks have emerged in other languages and exploded in popularity because they are natural vehicles for delivering programming-based research.

In its last few chapters, this book will place particular emphasis on the Axon project. An axon is part of a neuron, and the Axon framework supports the building and training of neural networks common in machine learning. The project is in its infancy but already has features such as gradient descent that are highly prized in other environments. All of the tools in the Nx toolbox come together to produce a credible machine learning foundation for Elixir.

Though Nx is in its infancy, it's not hard to immediately see the promise. Elixir already has ideal tools for the gathering, processing, and presentation of data. Nx can extend that language by using that data to decide what to do. This book will provide an overview of the Nx stack.

In this chapter, we're going to start with the basics in Nx, and that will take us to the tools you'll use to display, create, and manipulate tensors. We'll explore Nx within a Livebook for documenting numerical research with tensor data and prose. We'll use those notebooks to display and plot the contents of tensors. Next, we'll create tensors through multiple mechanisms. Finally, we'll manipulate them with operators, functions, and numerical definitions.

2.1. Document, Share, and Explore With Livebook

A Livebook presents data, prose, and code in a format that's friendly to browsers, the Elixir ecosystem, and file systems. Browser-friendly means users everywhere can read Livebooks. Elixir-friendly means Livebooks can run programs and interact with the reader in interesting ways. File-system friendly means Livebooks work with existing tools like code repositories and operating systems to save data.

Data science on platforms like Nx is a unique combination of scientific exploration, data, and code. That means experiments take the form of programs. Using notebooks, you can share findings. Livebooks even allow you to collaborate with a team.

Normally, you'll set up your own server to work with Livebook, and run it in production mode. You can find the project in the github livebook project. Once you load it, you can run through the introductory Livebook tutorials on Livebook itself, Elixir, and Distributed Elixir. We'll cover some of the Livebook highlights here.

Cells and Markdown

Livebook works in small units called cells. At least four types of cells are supported.

- Sections add a document section including a heading.
- Markdown allows the creation of marked up text, like this paragraph.
- Elixir cells allow inline Elixir, and displays the results.
- Input cells allow input controls that can be bound to variables.

As you work within a Livebook, you'll add a segment to your document with a control. If you hover your mouse between cells, you'll see a menu

that looks like this:

This list is likely to grow as Livebook adds new capabilities. You can add a cell by moving the mouse beneath one of the content cells and clicking one of the + buttons in the resulting hover menu.

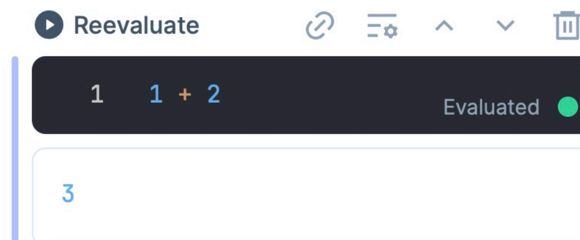
In this section, we'll build a Livebook together. If you want, you can follow along. Rather than show you screen shots of the contents of the Livebook, we'll show you code blocks with the input at the top and the output at the bottom, like this:

Sample Livebook Cell

```
1 + 2

> 3
```

We'll show that result in the place of a Livebook cell that looks like this:



The expression is at the top, and the result is at the bottom after the > character. If you'd like to follow along, enter the contents at the top, click the `evaluate` button, and Livebook will show the results at the bottom. If you ever change the contents of the cell, you'll see a * in the lower right. Simply click the `reevaluate` button to update the code contents, and the star will disappear. You can read about the buttons in the github Livebook project you downloaded. Most of the documentation is inside one of the Livebooks on the project page. Let's get back to the Livebook.

Rather than using the buttons, you can use shortcut keys to insert cells. `m` creates a markdown cell, and `n` creates an Elixir cell. To get a list of all of the shortcut keys, press the ? key. These keys will depend on whether you are actively editing a cell or not. The shortcut keys can insert, delete, and move cells. If you're editing Elixir code, the shortcut keys can even provide

code completion.

Run Elixir in a Cell

Elixir cells are pure Elixir code. Type Elixir in a cell. You can add variables you bind in one cell from another cell.

```
list = for x <- 1..3, y <- [:a, :b, :c], do: {x, y}

> [{1, :a}, {1, :b}, {1, :c},
 {2, :a}, {2, :b}, {2, :c},
 {3, :a}, {3, :b}, {3, :c}]
```

You can evaluate a cell at any time with a shortcut key, or by pressing the evaluate button in the upper left of any Elixir cell or the `evaluate` key sequence for your operating system. Remember, you can always press the ? key to get help.

Let's use the `list` binding and see what happens:

```
"Length: #{Enum.count(list)}"
> "Length: 9"
```

We use the value of `list` to present content. You can also access modules you've declared to build code in the very same way. Most of the time, declarations in one cell will be available to others. If you don't want this behavior, check the documentation for branching to create a cell in another process.

These basics will let you experiment with an isolated Livebook. If you're a programmer, you can probably spot a pretty big hole in this scenario. Programming projects are rarely isolated endeavors. For Livebook to be useful, we'll need to be able to, um, *mix* in our own projects. We'll dig into that capability next.

Mix Goodies

As we continue the leisurely stroll through the Livebook neighborhood, let's add a new section. Use the controls to add a new section to your book and mark it `Mix Tools`. As with any project, you might have dependencies to include. Livebook makes this easy with `Mix.install`.

Let's mix in the Nx dependency. Since there's no formal Hex project, we'll

need to add a git dependency. As you know, Hex is the package manager for Elixir. Git dependencies allow us to add a specific branch from git. We'll get the main branch with a sparse checkout. While we're at it, let's add the vega_lite and kino dependencies we'll need later for plotting, like this:

```
Mix.install([
  {:nx, "~> 0.1.0-dev", github: "elixir-nx/nx", branch: "main", sparse:
"nx"},
  :vega_lite,
  :kino
])
:ok
```

We just installed dependencies for Elixir's Nx and a few to support plots. VegaLite is a plotting library commonly used with Jupyter notebooks. Kino is a library for interactive widgets. We'll use both in the sections to come. The Livebook now has three dependencies and any other projects those three depend on. Now we can experiment with tensors and plots. Next, we'll add some additional tools to assist in the exploration of new APIs.

Like Livebook, IEx is another tool that offers Elixir experimentation, so it's sometimes helpful to make some IEx capabilities available in the context of a Livebook. It's easy to do. Just add a cell that makes the IEx helpers available, like this:

```
import IEx.Helpers
> IEx.Helpers
```

Now, you have access to the various IEx helper functions. You can get help on any new function, like this:

```
h(Nx.ceil())

> def ceil(tensor)
>
> Calculates the ceil of each element in the tensor.
>
> If a non-floating tensor is given, ...
```

Perfect. The various IEx tools make a potent combo with Livebook's features such as code completion and popup help. You can read about them by running the Livebook documentation notebook. Before moving on to tensors, there's one more Livebook topic to cover: plotting.

2.2. Plot Elixir Data in a Livebook

One of the most critical pieces of working with numerical data is visualization. Graphical libraries to handle plots are intricate, temperamental beasts. This work alone could occupy a whole team of programmers for years at a time. Fortunately, many existing libraries already exist. They already work with web-friendly JavaScript frameworks. Since Nx is a thin wrapper around tensors, it makes sense to wrap a tiny Elixir layer around an existing framework for plots.

The data science community has already accepted the VegaLite plot library and integrated it to notebooks. The Livebook engine has a library for rendering components called https://github.com/livebook-dev/kino. This new library separates the concepts of a rendering *interface* and a rendering *implementation*. All that remained for the Elixir community was to build a few Elixir datatypes for representing a plot, and integrate the plot in a Livebook using `Kino`. That's the purpose of VegaLite.

The `VegaLite` project combines three things: a struct representing a plot, some functions for creating it, and an Elixir protocol to render that plot using `Kino`. To work with VegaLite in a Livebook, all you need to do is build a struct with VegaLite's functions and leave the rendering to Livebook.

Let's see how it all works.

The `VegaLite` Struct Defines a Plot

To get started, let's add a new section to the livebook called `Plots`. Once you've done so, we'll add a few cells to explore the main `VegaLite` concepts. First, let's use the `IEx exports` helper to get information about the functions `VegaLite` supports, like this:

```
exports(VegaLite)

> __struct__/0  __struct__/1  concat/2  ...
```

For such a complex concept, there's not much here. As the introduction said, `VegaLite` is a wrapper around existing JavaScript tools. Those JavaScript functions will do the heavy lifting for turning the data and specifications we provide into an actual plot. Our job is to build an Elixir struct that actually describes our plot.

Let's look at a few of the functions we'll use. We'll start with a function to create a new `VegaLite` struct, the `new/1` function:

```
h(VegaLite.new())

> def new(opts \\ [])
> @spec new(keyword()) :: t()
> Returns a new specification wrapped in the VegaLite struct. ...

> See the docs (https://vega.github.io/vega-lite/docs/spec.html)
> for more details.
```

This argument takes a keyword list of options and returns a `t()`. When you're reading a type spec, think of `t()` as the main type for the given module. This is the foundational data we'll use to describe our plot. We'll use reducer functions to add more information to the plot, bit by bit.

One such reducer is `data_from_series/3`. This function will take a plot to set up the data. The first argument is a plot, and the second is a series of values. We'll look at this `series` argument in more detail in the example to come. Get the `help` page for `data_from_series`, like this:

```
h(VegaLite.data_from_series())

> def data_from_series(vl, series, opts \\ [])

>  @spec data_from_series(t(), Enumerable.t(), keyword()) :: t()

> Sets inline data in the specification. ...
```

The type spec tells the tale. It's a reducer, taking the telltale `t()` argument that specifies a `VegaLite` struct, and returning the same type, with the data added. You can drill down into other VegaLite reducers, but they all work the same way. When you're done, the Livebook will return a result. Since the result renders a Kino component, the VegaLite implementation of Kino's render function will arrange the data on the page in a format that the JavaScript version of VegaLite knows how to use.

You can see the common CRC pattern at play here. Start with an initial constructor, and then add to the initial specification a tiny bit at a time with a couple of reducers. Individual reducers will add data, specify the kind of mark to make, and mark each axis of the graph. When the specification is done, the pipeline is complete and the graph renders. Here's what a full program that renders part of a graph looks like:

```
alias VegaLite

time = 1..100
dist = fn x -> :math.pow(x, 2) end

VegaLite.new(width: 200, height: 200)
|> VegaLite.data_from_series(time: time, distance: Enum.map(time, dist))
|> VegaLite.mark(:line)
|> VegaLite.encode_field(:x, "time", type: :quantitative)
|> VegaLite.encode_field(:y, "distance", type: :quantitative)
```

This code creates a new `200 x 200` plot. Then, it creates data from a series with `time` and `distance` horizontal components, and designates it as a `line` plot. Next, it marks the `x` and `y` axes as numeric data with the appropriate slice of data, and leaves VegaLite to plot the chart described by the resulting data structure.

Let's dive down one more level. Let's use the `i` helper from IEx to drill into more details:

```
i(VegaLite.new())


> Term
  %VegaLite{spec: %{"$schema" => "https://vega.github.io/schema/vega-
lite/v5.json"}}
> Data type
  VegaLite
> Description
  This is a struct. Structs are maps with a __struct__ key.
> Reference modules
  VegaLite, Map
> Implemented protocols
  IEx.Info, Inspect, Kino.Render
```

This info page confirms that `VegaLite` is a struct, with only one JSON specification. That makes sense because `VegaLite` is a wrapper around a JavaScript library that renders a JSON object.

Next, look at the implemented protocols. `Keno.Render` is a supported protocol, and that's how the graph is rendered. You can read about the relationship between `Kino` and `Livebook` here.

With the exploration of plots complete, we're ready to shift toward tensors and the heart of Nx.