

Extracted from:

# Machine Learning in Elixir

Learning to Learn with Nx and Axon

This PDF file contains pages extracted from *Machine Learning in Elixir*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

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, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The  
Pragmatic  
Programmers



Your Elixir Source

# Machine Learning in Elixir

Learning to Learn with Nx and Axon



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](mailto:support@pragprog.com).

For international rights, please contact [rights@pragprog.com](mailto:rights@pragprog.com).

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. 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, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 979-8-88865-034-9

Encoded using the finest acid-free high-entropy binary digits.

Book version: B1.0—July 19, 2023

# Forecast the Future

---

In the previous chapter, you were introduced to the concept of recurrent neural networks for learning on sequential data. Specifically, you created and trained recurrent neural networks on text data using Elixir and Axon. You saw how recurrent neural networks are capable of learning relationships in natural language far easier than traditional feed-forward networks due to their built-in memory and inherent sequential operation.

Recurrent Neural Networks are well-suited for processing text. However, that's not the only thing they're good for. Anything with temporal nature presents challenges for traditional feed-forward networks and is a well-suited challenge for recurrent neural networks. One example of such data is *timeseries data*. Timeseries data is any collection of data indexed in time order. At each timestep, there are one or many observations. You can see how timeseries data lends itself naturally to working with recurrent neural networks.

In this chapter, you'll work with timeseries data in Elixir, Nx, and Axon. You'll learn about some of the challenges of working with timeseries data and a bit about why neural networks struggle so much with timeseries data compared to other approaches. You'll also train both a convolutional and recurrent neural network on a timeseries analysis problem, comparing the results and learning the benefits and drawbacks of each strategy.

## Predicting Stock Prices

Perhaps one of the most obvious applications of timeseries analysis is forecasting the direction of markets. Given enough historical data, you should be able to predict the future performance of a given equity or market, right? As you'll find out later in this chapter, the problem of forecasting markets is exceptionally difficult. If it was as easy as throwing a neural network at the problem, everybody would be doing it—and then it wouldn't be so easy any-

more. Despite the challenges, attempting to forecast stock prices is a good exercise in timeseries analysis, and a good demonstration of the pitfalls of putting too much faith in a model.

In this example, you'll be working with the historical stock data for 30 companies in the Dow Jones Industrial Average between 2006 and 2018. The data is available for download on Kaggle<sup>1</sup>. After you download the data, you'll see a collection of CSVs that contain information for the prices of individual stocks and for all of the stocks. To simplify the problem, you'll create a model that predicts the future stock prices for a single stock, in this case AAPL.

Start by firing up a new Livebook and adding the following dependencies:

```
Mix.install([
  {:explorer, "~> 0.5.0"},
  {:nx, "~> 0.5"},
  {:exla, "~> 0.5"},
  {:axon, "~> 0.5"},
  {:vega_lite, "~> 0.1.6"},
  {:kino, "~> 0.8.0"},
  {:kino_vega_lite, "~> 0.1.7"}
])
```

It's common to alias the VegaLite module to `vl`, so run the following code in a new cell:

```
alias VegaLite, as: vl
```

To simplify the process of working with the structured CSV data, you'll use *Explorer*, Elixir's DataFrame library. You'll use *Nx* and *EXLA* for numerical computing and acceleration, respectively, and you'll need *Axon* for the deep learning implementation. You'll also use *VegaLite*, *Kino*, and *Kino.VegaLite* for providing some functionality to visualize and summarize your dataset.

Before diving in, take some time to get familiar with the data. Start by loading it into a DataFrame using *Explorer*, like so:

```
csv_file = "all_stocks_2006-01-01_to_2018-01-01.csv"
df = Explorer.DataFrame.from_csv!(csv_file, parse_dates: true)
```

After running the code, you'll see the following output:

```
#Explorer.DataFrame<
  Polars[93612 x 7]
  Date string ["2006-01-03", "2006-01-04", "2006-01-05", ...]
  Open float [77.76, 79.49, 78.41, 78.64, 78.5, ...]
  High float [79.35, 79.49, 78.65, 78.9, 79.83, ...]
```

1. <https://www.kaggle.com/datasets/szrlee/stock-time-series-20050101-to-20171231>

```

Low float [77.24, 78.25, 77.56, 77.64, 78.46, ...]
Close float [79.11, 78.71, 77.99, 78.63, 79.02, ...]
Volume integer [3117200, 2558000, 2529500, ...]
Name string ["MMM", "MMM", "MMM", "MMM", "MMM", ...]
>

```

The dataset consists of opening, low, high, and closing prices on each trading day for a handful of stock tickers. You'll notice there's no intraday data and no *auxiliary information* aside from trading volume. Auxiliary information or side information is information you have access to outside of the target you're trying to predict. Rather than use side information, you're forced to make use only of past timesteps to predict future timesteps in an *autoregressive* manner. Autoregression means you're going to predict future values from existing values.

For the purposes of this problem, you'll pay attention to only a ticker's closing prices, so you can filter out the other irrelevant information:

```
df = Explorer.DataFrame.select(df, ["Date", "Close", "Name"])
```

After doing so, you'll see the following output:

```

#Explorer.DataFrame<
  Polars[93612 x 3]
  Date string ["2006-01-03", "2006-01-04", "2006-01-05", ...]
  Close float [79.11, 78.71, 77.99, 78.63, 79.02, ...]
  Name string ["MMM", "MMM", "MMM", "MMM", "MMM", ...]
>

```

Next, run the following code to get a visualization of the various stock tickers using VegaLite:

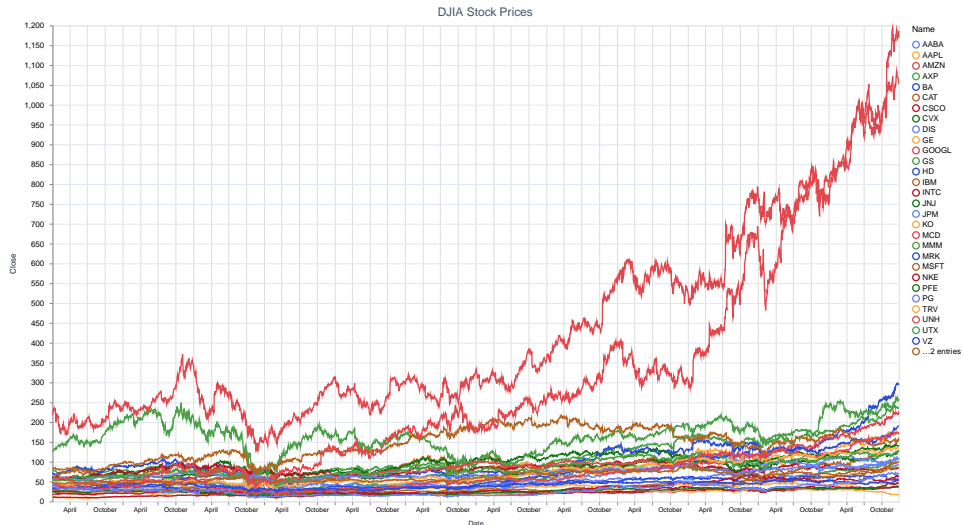
```

Vl.new(title: "DJIA Stock Prices", width: 640, height: 480)
|> Vl.data_from_values(Explorer.DataFrame.to_columns(df))
|> Vl.mark(:line)
|> Vl.encode_field(:x, "Date", type: :temporal)
|> Vl.encode_field(:y, "Close", type: :quantitative)
|> Vl.encode_field(:color, "Name", type: :nominal)
|> Kino.VegaLite.new()

```

And you'll see the following rendered image:





This image is, admittedly, a bit noisy. For this problem, you're only concerned with the price of the AAPL stock, so you can filter your data accordingly:

```
aapl_df = Explorer.DataFrame.filter_with(df, fn df ->
  Explorer.Series.equal(df["Name"], "AAPL")
end)
```

After you do, you'll see the following output:

```
#Explorer.DataFrame<
  Polars[3019 x 3]
  Date string ["2006-01-03", "2006-01-04", "2006-01-05", ...]
  Close float [10.68, 10.71, 10.63, 10.9, 10.86, ...]
  Name string ["AAPL", "AAPL", "AAPL", "AAPL", "AAPL", ...]
>
```

Now, regenerate your original plot with only AAPL data:

```
Vl.new(title: "AAPL Stock Price", width: 640, height: 480)
|> Vl.data_from_values(Explorer.DataFrame.to_columns(aapl_df))
|> Vl.mark(:line)
|> Vl.encode_field(:x, "Date", type: :temporal)
|> Vl.encode_field(:y, "Close", type: :quantitative)
|> Kino.VegaLite.new()
```

And you'll see the following rendered image:



Now it's time to start preparing the data for model training.

First, notice that your dataset consists of unnormalized stock prices. Remember that it's important to normalize data before you feed it into your neural network for training. Run the following code to normalize your DataFrame of AAPL stock prices:

```
normalized_aapl_df = Explorer.DataFrame.mutate_with(aapl_df, fn df ->
  var = Explorer.Series.variance(df["Close"])
  mean = Explorer.Series.mean(df["Close"])
  centered = Explorer.Series.subtract(df["Close"], mean)
  norm = Explorer.Series.divide(centered, var)
  ["Close": norm]
end)
```

You can replot your graph to verify that the pattern remains the same despite the normalization:



Next, you need to split your model between training and testing sets. In previous chapters, there was usually a clear delineation in input features and targets. That's not necessarily the case in this problem. In timeseries analysis, your goal is to predict either a single step or multiple steps in the future. You can make this prediction based on a range of historical inputs or based on the current timestep. In this example, you'll perform both single-step using a range of historical values.

You'll also need to divide your dataset into training and test sets. Start by creating the following window/3 function in a new module:

```
defmodule Data do
  def window(inputs, window_size, target_window_size) do
    inputs
    |> Stream.chunk_every(window_size + target_window_size, 1, :discard)
    |> Stream.map(fn window ->
      features =
        window
        |> Enum.take(window_size)
        |> Nx.tensor()
        |> Nx.new_axis(1)

      targets =
        window
        |> Enum.drop(window_size)
        |> Nx.new_axis(1)

      {features, targets}
    end)
  end
end
```

**end**

The `window/3` method takes an `Enumerable` or stream, an input window size, and a target window size. It then returns a new stream, where each element is a tuple of tensors with features and targets.

The features tensor is comprised of `window_size` prices from the last `window_size` days. The targets tensor is comprised of `target_window_size` prices from `target_window_size` days after the input window. Notice that for both features and targets you need to add a new axis. Recall from [Representing the World, on page ?](#), timeseries data is typically represented in three-dimensions with shape `{batch, timesteps, features}`. For this example, you only have a single feature—price—and you're only predicting a single feature: price.

Next, add the following `batch/2` function to your data module:

```
def batch(inputs, batch_size) do
  inputs
  |> Stream.chunk_every(batch_size)
  |> Stream.map(fn windows ->
    {features, targets} = Enum.unzip(windows)
    {Nx.stack(features), Nx.stack(targets)}
  end)
end
```

The `batch/2` method converts your input windows into batches of input windows by stacking features and targets on top of one-another. You can now use these methods to create new training and testing sets. But, how do you split up your data into training and test sets?

In the past, you randomly split your dataset into some percentage of training and testing data. You're able to shuffle split your dataset somewhat naively because your data does not have any temporal dependencies. With a time-series dataset, you have a bunch of potentially overlapping dependencies, which means you can leak information about your test set into the training process. Leakage isn't good, and it can result in overconfidence and deployment of bad models.

An alternative approach is to split your dataset temporally. For this example, you have 10 years worth of data, so you can take the first eight years as training data, and the last two years as testing data. While this approach is better, it's important to understand that there are still drawbacks. Certain time-series analysis problems, like stock price prediction, are often dependent on unseen macro windows. Your model may appear really good at predicting the price of a stock because the market was in the middle of a very consistent

bull run. However, if that fundamental macro trend changes for the test set, you'll have a garbage model.

A challenge with time-series analysis is that there are often *confounding variables* at play, and without access to that information, it can be difficult to get an accurate model. You need to be extremely careful when evaluating time-series models to ensure you've eliminated most possible sources of bias.

For this example, after looking at the plot of AAPL stock over the last 10 years, you can see the general trend: AAPL stock goes up. The macro trends between the training and testing set are relatively similar. However, that doesn't necessarily mean you'll end up with a model that's good at predicting stock prices for 2023 or 2024. One consideration when doing time-series analysis is that you will constantly need to re-train and update trends. The world is chaotic, and the future isn't easy to predict.

To split your data into training and test sets, run the following code:

```
train_df = Explorer.DataFrame.filter_with(normalized_aapl_df, fn df ->
  Explorer.Series.less(df["Date"], Date.new!(2016, 1, 1))
end)

test_df = Explorer.DataFrame.filter_with(normalized_aapl_df, fn df ->
  Explorer.Series.greater_equal(df["Date"], Date.new!(2016, 1, 1))
end)
```

After doing so, you'll see the following output:

```
#Explorer.DataFrame<
  Polars[503 x 3]
  Date date [2016-01-04, 2016-01-05, 2016-01-06, 2016-01-07, ...]
  Close float [105.35, 102.71, 100.7, 96.45, 96.96, ...]
  Name string ["AAPL", "AAPL", "AAPL", "AAPL", "AAPL", ...]
>
```

Now, convert your DataFrames to batches of windowed tensors using your data module by running the following code:

```
window_size = 30
batch_size = 32

train_prices = Explorer.Series.to_list(train_df["Close"])
test_prices = Explorer.Series.to_list(test_df["Close"])

single_step_train_data =
  prices
  |> Data.window(window_size, 1)
  |> Data.batch(batch_size)

single_step_test_data =
  prices
```

```
|> Data.window(window_size, 1)
|> Data.batch(batch_size)
```

You can confirm you've correctly created your datasets by taking samples from each. If you run the following code in a new cell:

```
Enum.take(single_step_train_data, 1)
```

you will see:

```
[
  {#Nx.Tensor<
    f32[32][5][1]
    [
      [
        [-0.027216043323278427],
        [-0.027200918644666672],
        [-0.02724125050008297],
        [-0.02710512839257717],
        [-0.027125295251607895]
      ],
      ...
    ]
  },
  {#Nx.Tensor<
    f32[32][1][1]
    [
      [
        [-0.026777423918247223]
      ],
      [
        [-0.026555592194199562]
      ],
      ...
    ]
  }
]
```

Now you have a dataset of training inputs and training outputs. Your inputs are a sample of 5 days of closing prices, with the target being a single closing price.

Now that you've built a dataset, it's time to start training some neural networks.