The Pragmatic Programmers

grox.io Learning

Programmer Passport

Julia ML with Flux

Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: Flux

Bruce A. Tate

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*.

Copyright © 2021 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: pending

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

Book version: B1.0—Monthname, yyyy

Contents

Change History

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.

B1.0: March 1, 2020

- A new quarter, a new release of Programmer Passport! The Flux framework will build on the work we've done in Julia and bring it into machine learning. In this release, we build the intellectual foundation you'll need to start solving problems. We'll introduce the core concepts of modeling, training, loss functions, and optimizing. Then we'll train a model so it can make predictions. Have fun!

CHAPTER 1

Build a Flux Classifier

The Julia Language has made significant inroads as a language that's ideal for all different kinds of high performance numerical computing. One of the most exciting use cases is machine learning (ML). The Flux library for Julia is the most popular for solving ML problems everywhere.

Whether you are trying to build a predictive model for house prices or provide the perfect medicine dosage based on historical performance and patient metrics, Flux is a good tool for the job, for reasons you'll see as we go through this module. The secret sauce is a mathematical technique called automatic differentiation (AD), a mathematical method for rapidly computing the rate of change. If you've not encountered these concepts before, you'll possibly find that much of the other available ML literature is impenetrable because of vocabulary based on dozens of years of math and computer science theory.

In this short book, you're going to take a whirlwind tour through some basic ML theory. Each chapter will walk you through one machine learning problem, along with the theory to support each one in layman's terms. We'll try to leave most of the jargon behind.

As far as iconic examples go, the Flux community provides a few core sample programs in the model zoo.[1] (Don't feed the models.) This first chapter will walk you through a reworked version of one of those problems, though not one that most ML experts focus on. We'll walk through the example that teaches a Flux model to play FizzBuzz,[2] a math game and program problem that places integers in several categories. Instead of using basic math rules, our model will learn to mimic a function without knowing about the rules inside.

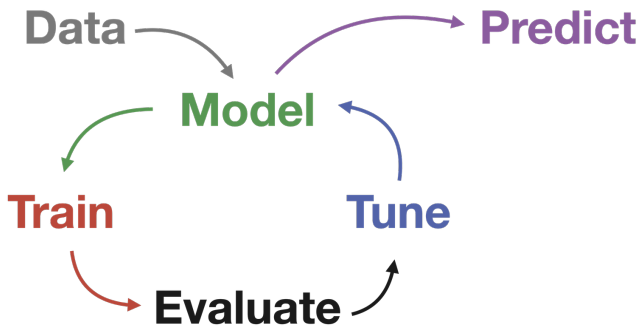
1. https://github.com/FluxML/model-zoo
2. http://rosettacode.org/wiki/FizzBuzz#Julia

Most Flux programs are dense with concepts, so we'll take three different passes through the same program. With each pass, we'll dive down into more detail. By the last pass through, we'll be breaking down each of the individual features. When you're done, you'll be able to read a basic Flux program and understand the different elements of the solution.

Let's get started.

How Machine Learning Works

Machine learning is the process of building *configurable functions* in *layers* and then *iteratively training them* to *make predictions.* That description is a bit abstract, so let's look at an image that breaks down a typical project into a multi-stage process:



This figure shows each of the stages of a machine learning process.

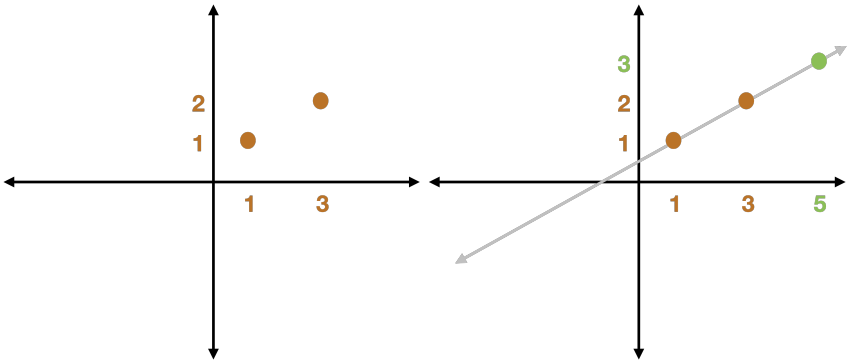
- Gather some real world *data* and prepare it as inputs into the *model.*
- Build a *model* composed of many *simple functions* with *configurable parameters*.
- Repeatedly *train* the parameters with real world data.
- Each training stage *evaluates* the model by calculating the distance between the prediction and real world data.
- The training stage repeatedly *optimizes* predictions by tweaking model parameters based on the evaluation stage.
- Once training is complete, make *predictions* using the updated model.

All Flux programs consist of these steps in some order. You'll collect and prepare data, make a model, then repeatedly train it with an evaluation function and an optimizer, and finally make predictions from the model.

Models Are Predictive Functions

Programming is about writing functions, and ML is about predicting the values of models composed of *nonlinear functions*. Let's take that statement apart.

A technique called linear regression[3] can solve linear models. These solutions find a best-fit line through a list of observations, as in the following figure.



On the left, the figure shows a collection of points. Then, on the right, you'll use linear regression to calculate a line through the points, and use those to make predictions like the green dot. The formula is not too complex, and it's not too expensive to solve computationally. Nonlinear problems are more difficult to solve.

Here's the problem. Many of the most important predictions a computer can make are nonlinear. Pandemics and populations both have elements of exponential growth. Language and image recognition systems are also nonlinear. That means you need a different strategy. Enter ML.

Neural Networks are Configurable Functions

At its core, ML solves *nonlinear equations* by composing a *complex model* from many *simple functions* with *many configurable parameters*. By repeatedly changing the parameters one tiny step at a time in a process called *training*, the model gets better and better at *predicting real-world results.*

Let's drill down into that paragraph a bit. The first part refers to *nonlinear equations in a configurable, complex model composed of simple functions.* Most often, that complex model relies on a concept called *neural networks*, or neural nets.

Neural Net Properties

Here's the bottom line. While these nets of functions might differ in their implementations, they are at once easy to represent and also make wonderfully complex predictions because of these four properties:

3. https://machinelearningmastery.com/linear-regression-for-machine-learning/

Layers : Neural nets are composed of *layers* tied together with functional *composition* (think |>, ∘ or Chain.)

Multiple inputs : Instead of taking primitive numeric arguments called scalars,[4], neural nets work on *multiple inputs* by taking matrices as arguments. The functions in a layer follow the rules of linear algebra.[5]

Configurable parameters : In the function 5x, 5 is a coefficient. In neural nets, coefficients are matrices of scalar values. Each scalar in the matrix can be configured, yielding a different prediction.

Activation functions : Activation functions are simple nonlinear functions. That means rather than representing straight line data, they can represent curves. Neural nets work mostly with linear functions, but they can represent nonlinear predictions by mixing in an activation function.

Don't sweat the details. You'll have plenty of time to absorb them throughout the chapter. For now, let's look at a slightly modified Julia program from the model zoo[6] with those properties in mind:

```
Chain(Dense(10, 5, relu), Dense(5, 2), softmax)
```

Believe it or not, this code represents a working neural net that demonstrates each of the four properties. Let's see how.

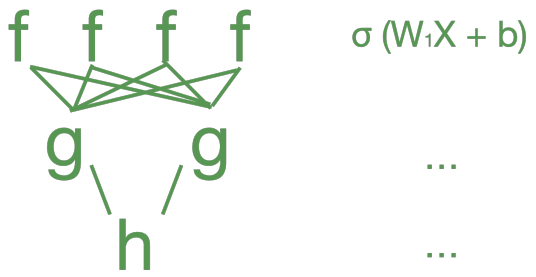
Dense builds one of the most common Flux layers. The Chain(...) statement composes one large function out of three smaller ones, including the two Dense layers we specify. The resulting function accepts *multiple inputs*. The Dense layers have *configurable parameters* called weights and biases that Flux can tweak to make predictions. The relu *activation function* is a nonlinear function that allows this network to make nonlinear predictions. The softmax function makes the resulting prediction friendlier by converting it to a probability. There you have it: a neural net.

Again, don't sweat the details. Just marvel in the fact that this network is a working one-line representation of a machine learning model. This brief example is like any other neural net. The simple representation is one side of the coin. Let's look at the other, models that can make complex predictions.

4. https://www.britannica.com/science/scalar
5. https://machinelearningmastery.com/gentle-introduction-linear-algebra/
6. https://github.com/FluxML/model-zoo/blob/master/other/flux-next/intro.jl

Complex Predictions

Just as physical nets have multiple dimensions, neural nets do too. One dimension comes from the multiple inputs, and the other comes from the layers. In a generic neural net, the result is something that looks like this:



The mathematical expression of the neural net is something like `f ∘ g ∘ h`, but when you use linear algebra to process the inputs and outputs, the result is more like the web of functions you see on the left. Each one of the individual layers is simple.

Look at the right hand side to see the mathematical representation of the functions making up a Flux `Dense` layer. The `W` and `b` are weights and biases that make up the configurable parameters to the neural net. A weight defines the *importance* of connections in the network, and a *bias* is a parameter impacting one layer. The Greek character is a sigmoid (σ), and is the activation function. Without a nonlinear function somewhere in the neural net, this model would only predict linear results because when larger functions are composed of smaller linear functions, the result must be linear. Eventually, you'll choose activation functions based on the training and predictive properties you want in your model.

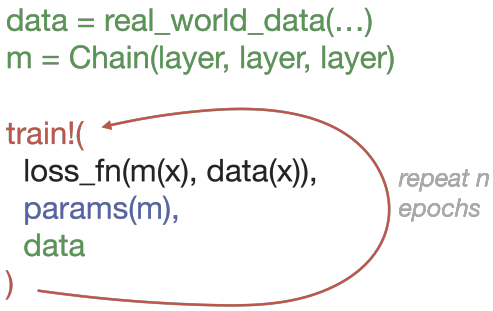
Now, let's drill down a bit into the configurable parameters of a neural net. *Training* is a an iterative loop where each iteration involves two smaller stages: *evaluate* a model's prediction against real world data, and *optimize* the parameters based on the results. The evaluation stage uses a function called a *loss function* to measure how much predictions differ from actual values, and Flux has several built in loss functions to choose from. The optimize stage tweaks the parameters made up of the weights and bias in each layer to improve the model. Flux uses AD to improve the process, making it both more understandable and more efficient.

Whew. That's abstract. Don't worry. We'll go through the process several times through the course of this material and you'll pick up those concepts and see

what's happening. This all seems abstract, so let's look at a full Flux program. Let's get a FizzBuzz on.

Anatomy of a Flux Program

Flux programs closely mirror the process you saw in the previous section. You'll prepare data, then build a model, and then train it. The training step will use a function to compare real world data with a model's prediction and then tweak the parameters with an evaluation step, as in the following figure.



This figure says it all. Get some data, build a model, and then train the model. When you're done, you'll have a function you can use to make predictions. Keep these steps in mind as we walk through an example, step by step, in the rest of the chapter.

The Rough Skeleton

FizzBuzz is a great program for understanding ML. It has well-understood data, can be expressed in a nonlinear function, and is composed of easily understood features. Some of these features depend on each other and some are independent. In many ways, ML predictions rely on relationships between features that may be difficult to see. Think about the impact of an additional bathroom on the price of a house in a a suburban neighborhood versus one downtown. Many of the same ideas in the FizzBuzz predictor from the model zoo will work in other applications as well.

Rather than reading through the whole program step-by-step, let's start with an overall skeleton with the function API intact but many of the details removed.

```
using ...

(X, y) = get_training_data()

build_model() = ...
function predict(x) ...
m = build_model()
```

```
loss(x, y) = ...
optimizer = ...
@epochs number_of_epochs train!(loss, params(m), [(X, y)], opt)

predict(...)
```

In short, we do these steps:

- Gather real world data for training
- Build a model and use it in a function to make predictions
- Build a function to objectively measure loss of accuracy
- Train the model using a loss function, training data, and an optimizer
- Use the trained model to make predictions

You'll see many different mechanisms to manage data, weave together models, calculate loss, and optimize, but these basic ideas will be present in every Flux program. As we go through this chapter, we'll slowly add detail until the model can predict the results of a FizzBuzz game without relying on the original function.

Let's add some flesh to these bones.

Build a Categorization Program

The FizzBuzz prediction program will use the typical Flux structure to eventually make predictions. We'll do so by gathering a corpus of training data which we'll use it to train a model. Initially, the model will make poor predictions, but each training step will improve the network a small bit until it makes accurate predictions. This description is pretty abstract, so let's look at a real program. First, we'll show the whole program with just a little commentary. Then, we'll break the program down into greater detail.

Gather Training Data

In a typical ML scenario, you'd often gather observations from the real world. A spam filter might need to collect known spam and non-spam emails with text that you might prepare by building word frequency tables. For FizzBuzz, you won't need to find and shape data. You'll be able to compute it instead. The function to compute FizzBuzz is pretty trivial:

```
using Flux: Chain, Dense, params, logitcrossentropy, onehotbatch, ADAM,
    train!, softmax

function fizzbuzz(x::Int)
    is_divisible_by_three = x % 3 == 0
    is_divisible_by_five = x % 5 == 0

    if is_divisible_by_three & is_divisible_by_five
```

```
        return "fizzbuzz"
    elseif is_divisible_by_three
        return "fizz"
    elseif is_divisible_by_five
        return "buzz"
    else
        return "else"
    end
end
```

It won't be enough to just print out a list of words and numbers, though. We'll need to shape the inputs and outputs so they can be used in our system.

Shape Training Data Using Features

Before our model can use data from fizzbuzz, we'll need to shape the data in a way that our model will understand. We'll break down the details later. For now, here's the program:

```
const LABELS = ["fizz", "buzz", "fizzbuzz", "else"];

features(x) = float.([x % 3, x % 5, x % 15])
features(x::AbstractArray) = hcat(features.(x)...)

getdata() = features(1:100), onehotbatch(fizzbuzz.(1:100), LABELS)
(X, y) = getdata()
```

You don't need to understand every line of code yet since we'll take several more passes through this function and explore the data it creates in more detail as the chapter evolves. The getdata function returns inputs and outputs. In both `X` and `y`, the data represents corresponding inputs and outputs. Each input is composed of features that improve the predictive power of the model. Each output is an encoded result. We'll talk more about the data specifics. For now, know the training data relates input features with real-world outputs.

Now, let's look at the model.

Build a Model and a Prediction Function

Models are *configurable functions*. The models are functions because you can call them with inputs, and they will generate outputs. The models are configurable because they have weights and biases that change to make different predictions. Remember, weights determine the *importance* of functions in the neural net. Training* is the process of changing those parameters to make better predictions, one small bit at a time. Here's the model.

```
fizzbuzz_model() = Chain(Dense(3, 10), Dense(10, 4))
function predict(x)
    prediction = m(features(x))
```