Deep Learning with Julia

Logan Kilpatrick

Logan Kilpatrick
The Julia Language
logan@julialang.org

https://deeplearningwithjulia.com

2021-11-30

Contents

1 About **3**

1.1 Getting Started 3

2 Preface **5**

2.1 Why Deep Learning? 5

2.2 Why does this book exist? 5

2.3 Acknowledgements 6

3 Why Julia **7**

3.1 Multiple Dispatch 7

3.2 Package Management 8

3.3 The Julia Community 10

4 Transfer Learning **11**

4.1 Pre-trained Models 11

4.2 Metalhead.jl 12

Appendix **15**

References **17**

1 About

Figure 1.1: Deep Learning with Julia

Deep Learning with Julia is a book about how to do various deep learning tasks using the Julia programming language and specifically the Flux.jl package. The intent of the book is to prove that serious deep learning can be done in Julia and that the ecosystem as a whole is ready for the spotlight.

1.1 Getting Started

All of the code and resources for this book are stored here on GitHub and deployed to https://deeplearningwithjulia.com.

This book is built via Books.jl[1] and is made possible by the Julia programming language (Bezanson et al., 2017) and pandoc[2].

[1] https://books.huijzer.xyz

[2] https://github.com/jgm/pandoc

2 *Preface*

The world is undergoing a radical shift. Machine Learning (and specifically Deep Learning) uses are being accelerated across every industry and domain. Humans have unlocked a significant new tool, the likes of which has not been seen since the software explosion took place in the early 2000's. But what does this mean for developers and those interested in working with these tools?

Like many field in the technology industry, there is a great deal of gate keeping in the Machine Learning community. Those in positions of power often make it seem as though using and understanding machine learning is reserved for those with PhD's and years of experience. This could not be farther from the truth. As the use of machine learning has increased, the barrier to entry have been slowly torn down. The result is that students with minimal programming and machine learning experience can now enter into the field and make advancements on the current state of the art.

In this book, we will touch on: - What the Julia Programming Language is and why we will be using it - What Deep Learning is - Various applications of Deep Learning

2.1 Why Deep Learning?

Why are we focusing on Deep Learning? How is that any different than machine learning? These are both great questions. We will delve more into the details in chapter 2, but the quick answer is that deep learning is a specific machine learning technique and the reason we want to focus on it is that many of the advancements as well as applications you have read about under the "AI" or "Machine Learning" title, are actually deep learning solutions under the hood. Use cases like Self Driving cars, digital voice assistants, recommendation engines (like on YouTube and Netflix) are all powered by Deep Learning.

2.2 Why does this book exist?

What was the point of writing this book? Currently, almost all deep learning practitioners use Python and most Deep Learning books focus on Python. Given the popularity of the language, this is a natural choice, especially given

the prevalence of high quality libraries like Pandas, Numpy, Tensorflow, Pytorch, etc. However, as the Julia programming language continues to grow and gain adoption, more and more users are coming and expecting a world class deep learning experience. While we have Flux.jl for deep learning in Julia, there are not currently any resources for learning about deep learning in Julia. Conversely, in the Python ecosystem, there are at least 4-5 foundational deep learning books which I personally used during my learning journey and I found to be excellent. The goal for this book is to show people that doing deep learning in Julia is a viable choice and more so than that, could actually come with significant advantages over other languages and frameworks.

2.3 Acknowledgements

I could write an entire book itself just going through all of the folks who have helped me get here. In general, this book, my career and life, are a product of support from a large group of amazing folks. From my parents and family, to teachers and professors, and especially the Julia community, without which, there would be no one reading this text. I will save you all from the rest of the sappy narrative but know that I appreciate each and every person who helped me get here.

3 *Why Julia*

If you have decided to pick up this book, you likely have heard or been told things about the awesome power of the Julia programming language. This chapter is dedicated for those who have not yet been convinced that Julia is the language of the future. If I don't need to convince you, please skip to the next chapter to dive into the fun. My personal hope is that one day soon, the Julia community will be large and mature enough that authors of Julia books need not include a "Why Julia" chapter. Until we get to that point, it is still worth it to talk about the benefits. Now back to Julia!

The Julia programming language was created in 2012 by a group of folks who believed that the scientific computing ecosystem could be better. They were fed up with MATLAB and Python because the former is not Open Source and pay to play while the latter is generally not performant enough to scale up in production environments. Researchers and programmers alike would generally use these tools for prototyping, but when it came time to deploy, they would be forced to rewrite their code in C++ or C in order to meet the performance thresholds required.

This phenomenon was coined as the "Two Language Problem" and Julia was created, in large part, to address it. After many years of hard work by Stefan Karpinski, Alan Edelman, Viral Shah, Jeff Bezanson, and enthusiastic contributors around the world, the language hit its 1.0 version release in 2018. The 1.0 release marked a huge milestone for the Julia community in terms of stability and the gave confidence to users that Julia would be along for the long haul.

In late 2021, Julia 1.6 was selected as the long term supported release. We will be using Julia 1.6 in this book so that the content will be as stable as possible for years to come.

Now that we have some historical context on Julia and why it was created, let us next move through some additional features which make Julia a natural choice for Deep Learning, Machine Learning, and more generally, science.

3.1 Multiple Dispatch

There is no one better to talk about the idea Multiple Dispatch and its use in Julia than Stefan Karpinski. In a 2019 JuliaCon talk titled "The Unreasonable

effectiveness of Multiple Dispatch" (https://youtu.be/kc9HwsxE1OY), Stefan went on to state that the Julia ecosystem has more code re-use than any ecosystem he has ever seen. Multiple Dispatch is the paradigm that allows this to happen. So what is Multiple Dispatch and why is it so unreasonably effective? For the latter point, I suggest watching Stefan's talk, there is no sense in re-stating what he already put eloquently. So back to the main question, what is multiple dispatch? The main idea here is that you can write multiple functions with the same name, which dispatch (or are called dynamically) depending on the types of the input arguments. This idea is not something necessarily unique to Julia, other languages have multiple dispatch or similar concepts. But the way in which it is used and implemented in Julia is the secret sauce. Let us now look at a quick example:

3.2 *Package Management*

Most people are unlikely to choose a programming language based on its package manager (or lack thereof). Despite this reality, the Julia package manager is one of those features that really makes me appreciate the language. Julia's package manager is extremely simple to work with and also enables much more reproducible code. Let us explore and see how this is the case:

There are two different fundamental ways of working with packages in Julia: via the REPL's Pkg mode and via the Pkg package. I will focus on using the REPL since it is extremely intuitive for new users. You can start by launching Julia in the terminal. Then, type `]` in which should take you into Pkg mode:

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.3 (2021-09-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

(@v1.6) pkg>
```

From here, one of the natural things to do is check what commands you can run in the package manager. To do this, type an `?` in and press enter/return.

You will see all the possible commands:

```
(@v1.6) pkg> ?
  Welcome to the Pkg REPL-mode. To return to the julia> prompt, either press
  backspace when the input line is empty or press Ctrl+C.
```

```
  Synopsis

  pkg> cmd [opts] [args]

  Multiple commands can be given on the same line by interleaving a ; between
  the commands. Some commands have an alias, indicated below.

  Commands

  activate: set the primary environment the package manager manipulates

  add: add packages to project

  build: run the build script for packages

  develop, dev: clone the full package repo locally for development
  ...
  ...
```

Some of the most common commands you will use are `add`, `activate`, `status` (or the shorthand `st`), and `remove` (or the shorthand `rm`). In this book, we will be using Flux.jl, so if you want to play around with the package, you can simply install it by typing `add Flux`.

After the install finishes, you can check your package environment by typing `status`:

```
(@v1.6) pkg> status
      Status `~/.julia/environments/v1.6/Project.toml`
  [587475ba] Flux v0.12.7
```

The package manager automatically shows the file which is managing the packages and their versions. In this case, it is `~/.julia/environments/v1.6/Project.toml`. As a general best practice, it is recommend to always use local environments instead of making changes to your main Julia environment. You can do this by activating a new environment.

```
julia> pwd()
"/Users/logankilpatrick"

shell> cd Desktop # type ';' to enter the shell mode from the REPL
/Users/logankilpatrick/Desktop

(@v1.6) pkg> activate .
  Activating new environment at `~/Desktop/Project.toml`
```

In the code above, you can see I started out in my main user folder. I then entered the shell mode by typing ; and used the change directory command to switch to my Desktop folder. From there, I did `activate .` which activates the current folder I am in. I can confirm this by typing `status`:

```
(Desktop) pkg> status
      Status `~/Desktop/Project.toml` (empty project)
```

and you can see the newly created project is empty.

Hopefully these examples give you a sense of how easy to user and powerful the Julia package manager is. You can read more about working with packages from the Pkg.jl docs: https://pkgdocs.julialang.org/v1/getting-started/.

3.3 The Julia Community

I would be remiss if I did not mention the Julia community itself as one of the core features of the ecosystem. Without such an incredible community, Julia would not be what it is today. But what makes the Julia community so great you might ask?

4 *Transfer Learning*

Transfer learning is one of the most useful and underrated deep learning tools. It allows us to take a model which was trained on a large data set, and "fine tune" it (more on what that means later) to work well for our specific use-case. It gets its name from the idea of learned information being transferred from one use case to another.

You might be asking yourself, why is this so powerful? Well, during the learning process, a deep learning model has to figure out things like what an edge looks like in the case of computer vision. This might seem like an abstract idea but it is a critical step for the model to learn how to distinguish between multiple objects. When we use transfer learning, many of these ideas have already been learned by the existing model we are starting with. This means that we do not need to spend as long training since already know some info about objects, even though the pre-trained model might never has seen data similar to the data you will be feeding into it.

Here is a simple example to try and illustrate why transfer learning works so well: imagine you are trying to teach someone what a car is. This person has never seen a car nor knows what it does. This person has seen a bicycle before and in fact uses one everyday. You can now explain to the person what a car is in terms of how it relates to a bike. The transfer learning process is much like this. Use the existing info that the model has learned and build off of that for some specific situation.

4.1 Pre-trained Models

The way in which we do transfer learning in the context of deep learning is with pre-trained models. But what is a pre-trained model? In general, we are referring to a model which has been trained on a specific data set. We will be exploring transfer learning in the context of computer vision so this means the model saw many images, each with a label which says what it is, and over time the model learned to correctly say the label given the image. There are a lot of different ideas going on here: computer vision, datasets, transfer learning, and more. *If any of this is not making sense, that is totally expected, there is both a lot of jargon as well as many new ideas being introduced. Try to stay focused on the high level and we will come back to many of these topics in more detail.*

Now that we know the high level idea of transfer learning, let us dive into a real example using Flux. To give some context, in other machine learning frameworks like PyTorch, the pre-trained models are built right into PyTorch itself. In the Flux ecosystem however, the pre-trained models live in a package called Metalhead.jl[1]. Metalhead is built to work with the Flux ecosystem so you do not need to work about writing compatibility issues.

[1] https://github.com/FluxML/Metalhead.jl

4.2 *Metalhead.jl*

Let us start out by installing Metalhead in the package manager by doing `add Metalhead`. Then we can type `using Flux, MetalHead` into a Julia terminal session. Metalhead provides a number of different model types like `resnet`, `vgg`, and more. Over the course of your deep learning experience, you will become more familiar with these model types as they represent some of the most common models for transfer learning. Before we dive into actually using pre-trained models, we will first take a quick look at the model structure which we get from Metalhead and Flux.

```
julia> model = ResNet50(pretrain=false)
ResNet(
  Chain(
    Chain(
      Conv((7, 7), 3 => 64, pad=3, stride=2),  # 9_472 parameters
      BatchNorm(64, relu),              # 128 parameters, plus 128
      MaxPool((3, 3), pad=1, stride=2),
      Parallel(
        Metalhead.var"#18#20"(),
        Chain(
          Conv((1, 1), 64 => 64, bias=false),  # 4_096 parameters
          BatchNorm(64, relu),          # 128 parameters, plus 128
          Conv((3, 3), 64 => 64, pad=1, bias=false),  # 36_864 parameters
          BatchNorm(64, relu),          # 128 parameters, plus 128
          Conv((1, 1), 64 => 256, bias=false),  # 16_384 parameters
          BatchNorm(256),               # 512 parameters, plus 512
        ),
        Chain(
          Conv((1, 1), 64 => 256, bias=false),  # 16_384 parameters
          BatchNorm(256),               # 512 parameters, plus 512
        ),
      ),
      ...
      ...
      ...
    Chain(
      AdaptiveMeanPool((1, 1)),
      Flux.flatten,
      Dense(2048, 1000),                # 2_049_000 parameters
```

```
    ),
  ),
)         # Total: 162 trainable arrays, 25_557_096 parameters,
          # plus 106 non-trainable, 53_120 parameters, summarysize 97.749 MiB.
```

Now that is a lot to take in (even though most of the model has been omitted for space reasons), but at a high level, it represents the structure of the `ResNet↪50` model as defined in Flux. Notably, the `50` in `ResNet50` comes from the fact that the model has 50 layers. There are other ResNet like models with different numbers of layers but with the same overall structure.

Appendix

This is the appendix.

References

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98.