Deep Learning with Flux.jl

JuliaAcademy

Instructor Info —

Matt Bauman

Dr. Matt Bauman attained his PhD at the University of Pittsburgh studying neural engineering.

He has extensively used parallel computing and machine learning through his work there, at the University of Chicago's Data Science for Social Good Fellowship, and now as a Senior Research Scientist at Julia Computing.

Course Curriculum

- Course duration: 96 min total
- Flux Basics
- Using Fux
- Prereq: Fundamentals of Machine Learning

Overview

In this course, we build upon the previous course, Fundamentals of Machine Learning. We explore Julia's powerful Deep Learning package, Flux.jl, and use its high-level APIs and frameworks to define our ML toolchain and pipeline in easily. Over the course of 4 video lectures, we experiment with multi-level, multi-layer neural network architectures in an interactive manner via Jupyter Notebooks.

Course Summary

Flux Basics

Flux contains many predefined ML features, including activation functions e.g. Simgoid function via σ, loss functions e.g. `mse(model(x), y)`, and neurons e.g. `model = Dense(2, 1,`σ`)`. Activation functions can be directly performed on the parameters of `model`, e.g. via σ `.(model.W*x+model.b)`. `CSV.jl` and `DataFrame.jl` are 2 commonly used packages for organising and loading data. Flux tracks the history of operations performed to compute the model loss using `Flux.Tracker` such that the weights can be properly backpropagated. The Gradient Descent update rule is computed by `model.W/b.data .= model.W/b.grad *`α, s.t. α is the learning rate. Instead of implementing the above from scratch, Flux provides a high-level API to conduct training and optimisation, as per the snippet below. To visualise results, one can create contour plots using the `Plots.jl` library.

```
using Flux
using Flux: @epochs

model = Dense(2, 1, σ)
loss(x,y) = mse(model(x), y)
optimiser = ADAM(1e-04)
epoch_num = 10

@epochs epoch_num train!(loss, params(model), trainset, optimiser)
```

Using Flux

To truly explore the capabilities of Flux, we extend the binary classification example of the previous course to comprise 3 classes. This requires input $\boldsymbol{x}$, output $\boldsymbol{y}$ and bias $\boldsymbol{b}$ to all be vectors of length n (number of classes), and weight $\boldsymbol{W}$ to be a matrix. Instead of using a binary scalar to represent predictions, we know use one-hot vectors, where each element is set to `false` except the one corresponding to the correct class. A neuron with a sigmoid activation function can be represented as follows

$$\boldsymbol{\sigma}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{b}) = \begin{bmatrix} \sigma^{(1)} \\ \vdots \\ \sigma^{(n)} \end{bmatrix} = \frac{1}{1+e^{(-\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})}} \tag{1}$$

As we are no longer doing binary classification, the output function (hence decision boundary) needs to change. The softmax function $P(\boldsymbol{y}=\boldsymbol{j}|\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^T\boldsymbol{W}_j}}{\sum_{n=1} e^{\boldsymbol{x}^T\boldsymbol{W}_n}}$ – which computes the probability that each samples falls into each possible class – is used. To model increasingly complex data distributions, we need to increase model complexity by adding non-linear hidden layers. We can do this in Flux by chaining layers together `Chain(Conv((2,2),1=>16,relu), MaxPool((2,2),stride=(2,2)), x->reshape(x,:,size(x,4)), Dense(1028,256, tanh) Dense(256,64,relu), Dense(64,3), softmax)`. The cross entropy loss function can be used for better optimisation. By vectorising our operations, we can perform acceleration by passing matrix multiplications to be performed in parallel by GPUs.

Working with MNIST Data

We apply our knowledge to the MNIST dataset to recognise `(28,28)` greyscale handwritten digits from 1 to 9. It can be easily loaded with `Flux.data.MNIST`. For pre-processing, we generate a vector of floats via `vec(Float32.(img))`, format labels as onehot-vectors, and partition images into mini-batches. We can follow procedures in Figure 1 for training. To gauge model performance, we compare training & testing loss. Because we actively fit the trainset and the testset is unseen, we expect larger loss for the latter. If the gap is significant, the model has overfit and fails to generalise to the entirety of the dataset. This severely lowers the test accuracy and can be countered via techniques – cross validation, adding dropout layers, increasing size of training set, and decreasing model complexity. If the reverse is true, it is highly likely that the code is faulty.