Les Cahiers du GERAD ISSN: 0711–2440

`FluxNLPModels.jl` and `KnetNLPModels.jl`: connecting deep learning models with optimization solvers

F. Rahbarnia, P. Raynaud

G–2023–25

July 2023

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2023
– Bibliothèque et Archives Canada, 2023

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2023
– Library and Archives Canada, 2023

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

`FluxNLPModels.jl` and `KnetNLPModels.jl`: connecting deep learning models with optimization solvers

Farhad Rahbarnia [a]

Paul Raynaud [a, b]

[a] *GERAD & Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Qc), Canada, H3T 2A7*

[b] *GSCOP, 38031 Grenoble, France*

farhad.rahbarnia@polymtl.ca
paul.raynaud@polymtl.ca

July 2023
Les Cahiers du GERAD
G–2023–25

Abstract : This paper presents `FluxNLPModels.jl` and `KnetNLPModels.jl`, new Julia packages enabling a neural network, modelled with either Flux.jl or Knet.jl, to be trained by solvers from JuliaSmoothOptimizers.

Keywords: Julia, numerical optimization, nonlinear optimization, deep-neural network

Résumé : Cet article présente `FluxNLPModels.jl` et `KnetNLPModels.jl`, des nouveaux modules Julia permettant à des réseaux de neurones, définis par Flux.jl ou Knet.jl, d'être entraînés par les méthodes de minimisation développées dans JuliaSmoothOptimizers.

Mots clés : Julia, optimisation numérique, optimization non-linéaire, réseau de neurones profond

Acknowledgements: This work has been supported by the NSERC Alliance grant 544900–19 in collaboration with Huawei-Canada and NSERC PGS–D.

1 Summary

`FluxNLPModels.jl` and `KnetNLPModels.jl` are Julia (Bezanson et al., 2017) modules, part of the JuliaSmoothOptimizers (JSO) ecosystem (Migot et al., 2021). They are designed to bridge the gap between JSO optimization solvers and deep neural network modeling frameworks, specifically Flux.jl (Innes et al., 2018) and Knet.jl (Yuret et al., 2020).

Both Flux.jl and Knet.jl allow users to model deep neural network architectures and combine them with a loss function and a dataset from MLDataset (L. and S., 2016). These frameworks support various usual stochastic optimizers, such as stochastic gradient (Lecun et al., 1998), Nesterov acceleration (Nesterov, 1983), Adagrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2017).

FluxNLPModels.jl and KnetNLPModels.jl adopt the triptych of architecture, dataset, and loss function to model a neural network training problem as an unconstrained smooth optimization problem conforming to the NLPModels.jl API (Orban et al., 2020). Consequently, these models can be solved using solvers from, e.g., JSOSolvers (Orban et al., 2021a), which include gradient-based first and second-order methods. Limited-memory quasi-Newton methods (Byrd et al., 1994; Lu, 1996; Liu and Nocedal, 1989) can be used transparently by way of NLPModelModifiers (Orban et al., 2021b). Contrary to usual stochastic optimizers, all methods in JSOSolvers enforce decrease of a certain merit function.

While it is possible to write and integrate solvers directly into Flux.jl or Knet.jl, separating them into standalone packages offers advantages in terms of modularity, flexibility and ecosystem interoperability. Leveraging existing packages within the Julia ecosystem allows developers to tap into a broader range of optimization solvers.

We hope the decoupling of the modeling tool from the optimization solvers will allow users and researchers to employ a wide variety of optimization solvers, including a range of existing solvers not traditionally applied to deep network training such as R2 (Birgin et al., 2017a,b).

2 Statement of need

Flux.jl and Knet.jl, as standalone frameworks, do not have built-in interfaces with general optimization frameworks like JSO. However, they offer convenient modeling features for defining neural network architectures. These frameworks provide pre-defined neural layers, such as dense layers, convolutional layers, and other complex layers. Additionally, they allow users to initialize the weights using various methods, such as uniform distribution.

Both offer a wide range of loss functions, e.g., negative log likelihood, and provide the flexibility for users to define their own loss functions according to their specific needs. These frameworks enable efficient evaluation of the sampled loss and its derivatives, as well as the neural network output, on both CPU and GPU. This flexibility allows the weights to be represented as either a Vector (for CPU) or a CUVector (for GPU), with support for multiple floating-point systems. They facilitate the definition of minibatches as iterators over the dataset, enabling efficient batch processing during training.

The solvers in JSOSolvers are deterministic. However, the integrated callback mechanism allows the user to change the training minibatch and its size at each iteration, which effectively produces stochastic solvers. Finally, Flux.jl and Knet.jl offer convenient methods for evaluating the accuracy of the trained neural network on the test dataset.

The FluxNLPModels.jl and KnetNLPModels.jl modules have been developed to expand the range of optimization methods available for training neural networks defined with Flux.jl and Knet.jl. They leverage the tools provided by JSO to enable the use of a broader set of optimization techniques without the need for users to reimplement them specifically for Flux.jl or Knet.jl. This integration allows researchers and users from the deep learning community of Julia to benefit from advances in optimization. On the other side, researchers in optimization will benefit from advances in modeling network developed by either of the Flux.jl and Knet.jl communities.

3 Training a neural network with JuliaSmoothOptimizers solvers

In the following section, we illustrate how to train a LeNet architecture (Lecun et al., 1998) using JSO solvers. We assume that a `LeNet` model is defined in Flux.jl and the MNIST dataset (Lecun et al., 1998) from MLDataset has been downloaded, loaded, and minibatch loaders have been created as `train_loader, test_loader`. To view an example, please refer to LeNet-MNIST example from the 'example' folder of FluxNLPModels.jl.

To cast the LeNet model as an FluxNLPModel, one needs to pass the model that was defined in Flux, the loss function, as well as the train and test data loaders. Flux.jl allows flexibility to define any loss function we need. We will use the built-in `Flux.logitcrossentropy`.

```
using FluxNLPModels

LeNetNLPModel = FluxNLPModel(LeNet,
train_loader,
test_loader;
loss_f = Flux.logitcrossentropy)
```

After completing the necessary steps, one can utilize a solver from JSOSolvers to minimize the loss of LeNetNLPModel. These solvers have been primarily designed for deterministic optimization. In the case of FluxNLPModel.jl (and KnetNLPModels.jl), the loss function is managed to ensure its application to sampled data. However, it is essential to modify the training minibatch between iterations. This can be accomplished by leveraging the callback mechanism incorporated in JSOSolvers. This mechanism executes a pre-defined callback at the conclusion of each iteration. For more comprehensive information, we refer the reader to the JSOSolvers documentation.

In the following code snippet, we demonstrate the execution of the R2 solver with a `callback` that changes the training minibatch at each iteration:

```
using JSOSolvers

max_time = 300. # run at most 5min
callback = (LeNetNLPModel,
solver,
stats) -> FluxNLPModels.minibatch_next_train!(LeNetNLPModel)

solver_stats = R2(LeNetNLPModel; callback, max_time)
test_accuracy = FluxNLPModels.accuracy(LeNetNLPModel)
```

Another choice to train `LeNetNLPModel` is the LBFGS solver with linesearch:

```
solver_stats = lbfgs(LeNetNLPModel; callback, max_time)
```

To exploit any non-convexity present in `LeNetNLPModel`, an `LSR1` approximation of the Hessian which can be employed and fed into the `trunk` solver, which utilizes a trust-region method with a backtracking linesearch. To integrate the LSR1 approximation and trunk into the training process, the code can be modified as:

```
using NLPModelsModifiers # defines LSR1Model
```

```
lsr1_LeNet = NLPModelsModifiers.LSR1Model(LeNetNLPModel)
callback_lsr1 =
(lsr1_LeNet, solver, stats) -> FluxNLPModels.minibatch_next_train!(
lsr1_LeNet.model
)
solver_stats = trunk(lsr1_LeNet; callback = callback_lsr1, max_time)
```

The same example exists also for KnetNLPModels.jl, and we refer the reader to the LeNet training documentation for more details.

References

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

E. G. Birgin, J. Gardenghi, J. M. Martínez, S. A. Santos, and P. L. Toint. Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order regularized models. Mathematical Programming, 163:359–368, 2017a.

E. G. Birgin, J. L. Gardenghi, J. M. Martínez, S. A. Santos, and Ph. L. Toint. Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order regularized models. Mathematical Programming, 163(1):359–368, May 2017b.

R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. Math. Program., 63(1):129–156, Jan 1994.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res., 12(null):2121–2159, jul 2011.

M. Innes, E. Saba, K. Fischer, D. Gandhi, M. Concetto Rudilosso, N. Mariya Joy, T. Karmali, A. Pal, and S. V. Fashionable modelling with flux. Computing Research Repository, abs/1811.01457, 2018.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

C. L. and C. S. Mldatasets.jl:, 2016.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, Nov 1998.

D. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. Math. Program., 45(1-3):503–528, 1989.

X. Lu. A computational study of the limited memory SR1 method for unconstrained optimization. University of Colorado, Boulder, 1996.

T. Migot, D. Orban, and A. S. Siqueira. The JuliaSmoothOptimizers ecosystem for linear and nonlinear optimization, 2021.

Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. Proceedings of the USSR Academy of Sciences, 269:543–547, 1983.

D. Orban, A. S. Siqueira, and contributors. Nlpmodels.jl: Data structures for optimization models, July 2020.

D. Orban, A. S. Siqueira, and contributors. Jsosolvers.jl: Juliasmoothoptimizers optimization solvers, March 2021a.

D. Orban, A. S. Siqueira, and contributors. NLPModelsModifiers.jl: Model modifiers for nlpmodels. https://github.com/JuliaSmoothOptimizers/NLPModelsModifiers.jl, March 2021b.

D. Yuret, C. Gümeli, C. Lucibello, E. Onat, E. Akyürek, E. Emre Yurdakul, E. Ünal, E. Yolcu, E. Berk, E. Dayanık, I. Kesen, K. Xu, M. Melike Softa, M. Innes, O. Kuru, O. Arkan Can, O. Kırnap, P. Nguyen, R. Donner, T. Besard, and Z. Shiwei. Knet.jl: Koç university deep learning framework., 2020.