Summary

Deep neural network(DNN) modules, similar to the Julia module (ref) Knet.jl, are generally standalone modules whose provide:

- deep neural networks modelling;
- support standard training and test datasets (from MLDatasets.jl in Julia);
- several loss-functions, which may be evaluated from a mini-batch of a dataset;
- evaluate the accuracy of a neural network from a test dataset;
- GPU support of any operation performed by a neural network;
- state-of-the-art optimizers: SGD, Nesterov, Adagrad, Adam (refs), which are sophisticate stochastic line-search around first order derivatives of the loss-function.

Due to their design focused on machine learning, those modules lack interfaces with pure optimization frameworks such as JSOSolver (ref).

KnetNLPModels.jl tackles this issue by enabling wrapping DNN into unconstrained models. It inhernat and implement most, if not all, Knet's interfaces, such as:

- standard training and test datasets
- its lost functions
- ability to divide datasets into user-defined-size minibatches
- support GPU/CPU interface

KnetNLPModel benefits from the JuliaSmoothOptimizers ecosystem and is not limitted to the Knet solvers It has access to:

- JSOSolvers.jl optimizers, which train the neural network by considering the weights as variables;
- augmented optimization models such as quasi-Newton models (LBFGS or LSR1).

Example -- name to change

The following example (ref) covers traing a DNN model on MNIST (ref) data sets. It includes loading the data, defining DNN model, here Lenet-5 (ref), setting the mini-batches, and traing using R2 solver from JSOSolvers.

The main step is to transfer a Knet model to KnetNLP model, this can be achived by:

```
# LeNet is defined model for Knet
LeNetNLPModel = KnetNLPModel(
        LeNet;
        data_train = (xtrn, ytrn),
        data_test = (xtst, ytst),
        size_minibatch = minibatchSize,
    )
```

`KnetNLPModel` takes **LeNet**, a small DNN model defined in Knet using `Chainnnll`, train and test data, as well as user-defined batchsize.

Once the KnetNLP model is created, solvers from JSOSolver can be used. Here is an example using R2 solver

```
solver_stats = R2(
        modelNLP;
        callback = (nlp, solver, stats, nlp_param) ->
            cb(nlp, solver, stats, stochastic_data),
    )
```

For more information on R2 solver, refere to (ref)

To change the mini-batch data and update the epochs, a callback method can be constructoed and passed on to the R2 solver.

```
function cb(
    nlp,
    solver,
    stats,
    data::StochasticR2Data,
)
    # Max epoch
    if data.epoch == data.max_epoch
        stats.status = :user
        return
    end
    data.i = KnetNLPModels.minibatch_next_train!(nlp)
    if data.i == 1   # once one epoch is finished
        # reset
        data.grads_arr = []
        data.epoch += 1
        acc = KnetNLPModels.accuracy(nlp) # accracy of the
        train_acc = Knet.accuracy(nlp.chain; data =
nlp.training_minibatch_iterator) #TODO minibatch acc.
    end
end
```

We used a stuct to pass on different values and keep track of the accracy durong the training.

To check the accracy of the train or test data, use:

```
train_acc = Knet.accuracy(nlp.chain; data = nlp.training_minibatch_iterator) #TODO
minibatch acc.
```

To allow use of GPU we need the `Knet.array_type` to be set, we can achive that using:

```
if CUDA.functional()
    Knet.array_type[] = CUDA.CuArray{T}
else
    Knet.array_type[] = Array{T}
end
```

Statement of need

Acknowledgements

This work has been supported by the NSERC Alliance grant 544900-19 in collaboration with Huawei-Canada

References