RSO: Fitting a neural network by optimizing one weight at a time

STMO

Heesoo SONG

IMPORTANT!! Install MLDatasets packages first:
Pkg.add("MLDatasets")

```
using Combinatorics ✓, Plots ✓, Statistics ✓, Distributions ✓, MLDatasets ✓,
Flux ✓, CUDA ✓, Test ✓
```

```
using Flux ✓: Data.DataLoader
```

```
using Flux ✓: onehotbatch, onecold, crossentropy
```

```
using Flux ✓: @epochs
```

```
using Base: @kwdef
```

```
using Flux.Losses ✓: logitcrossentropy
```

0. Introduction

Research Goal

> **(1)** Propose RSO (random search optimization), a new weight update scheme for training deep neural networks, and **(2)** compare its accuracy to backpropagation.

RSO: Random Search Optimization

RSO is a new weight update algorithm for training deep neural networks which explores the region around the initialization point by sampling weight changes to minimize the objective function. The idea is based on the assumption that the initial set of weights is already close to the final solution, as deep neural networks are heavily over-parametrized. Unlike traditional backpropagation in training deep neural networks that involves estimation of gradient at a given point, **RSO is a gradient-free method that searches for the update one weight at a time with random sampling**. The formal expression of the RSO update rule is as following:

$$w_{i+1} = \begin{cases} w_i, & f(x, w_i) \leq f(s, w_i + \Delta w_i) \\ w_i + \Delta w_i, & f(x, w_i) > f(s, w_i + \Delta w_i) \end{cases}$$

, where Δw_i is the weight change hypothesis.

According to the paper, there are some **advantages** in using RSO over using backpropagation(SGD).

- RSO gives very close classification accuracy to SGD in a very few rounds of updates.
- RSO requires fewer weight updates compared to SGD to find good minimizers for deep neural networks.
- RSO can make aggressive weight updates in each step as there is no concept of learning rate.
- The weight update step for individual layers is not coupled with the magnitude of the loss.
- As a new optimization method in training deep neural networks, RSO potentially lead to a different class of training architectures.

However, RSO also has a **drawback** in terms of computational cost. Since it requires updates which are proportional to the number of network parameters, it can be very computationally expensive. The author of the paper however suggests that this issue can be solved and could be a viable alternative to back-propagation if the number of trainable parameters are reduced drastically as in [3].

In the following sections, we will reproduce the RSO function and compare its classification accuracy to the classical backpropagation method (SGD). In addition, we will have a look how the RSO algorithm performs in different models and batch sizes. To save your time, we used a simple model

with one convolutional layer rather than the original model from the paper which was comprised of 6 convolutional layers.

1. Construct RSO function

1-1. Parameters

RSO function is constructed by following the pseudocode provided in the paper. The variables used in the pseudocode are explained below:

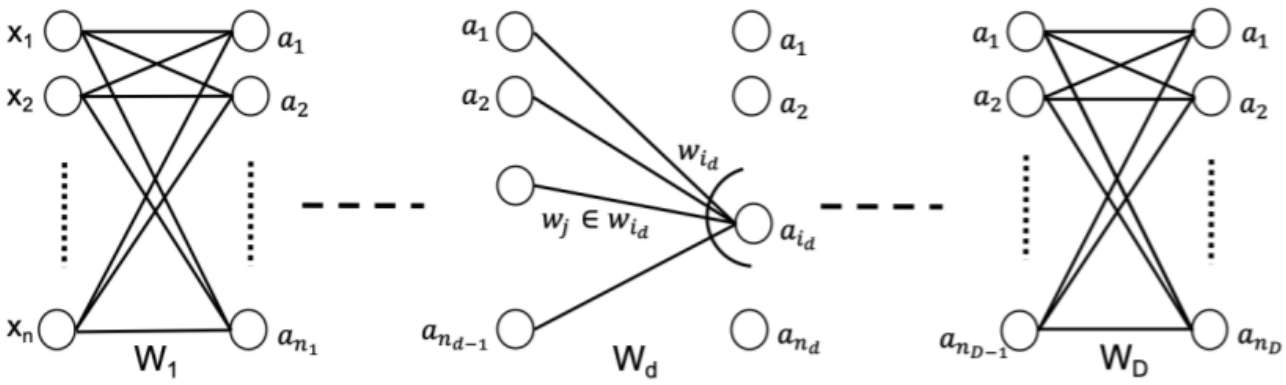


Figure 2: Notation for variables in the network architecture are shown in this figure.

- $W = \{W_1, \ldots, W_d, \ldots, W_D\}$ = Weight set of layers
- $W_d = \{w_1, \ldots, w_{i_d}, \ldots, w_{n_d}\}$ = Weight tensors of layer d that generates an activation set $A_d = \{a_1, \ldots, a_{i_d}, \ldots, a_{n_d}\}$
- w_{i_d} = a weight tensor that generates an activation a_{i_d}
- w_j = a weight in wid

1-2. Pseudocode

Algorithm 1: Random Search Optimization

Input : $\mathcal{X}$,$\mathcal{L}$,C,$W = \{W_1, W_2 ... W_D\}$,
where $W_d = \{w_1, w_2.., w_{i_d}, ..w_{n_d}\}$
Output : W

$\mathcal{X} \leftarrow \frac{\mathcal{X} - \Sigma X}{\sigma(\mathcal{X})}$
for $W_d \in W$ **do**
  for $w_j \in w_{i_d}$, *where* $w_{i_d} \in W_d$ **do**
    $w_j \leftarrow \mathcal{N}(0, \sqrt{2/|w_{i_d}|})$
  end
  $\sigma_d \leftarrow \sigma(W_d)$
end
for $c = 1$ *to* C **do**
  $d \leftarrow D$
  while $d > 0$ **do**
    for $w_j \in w_{i_d}$, *where* $w_{i_d} \in W_d$ **do**
      $x, l \sim \mathcal{X}, \mathcal{L}; \Delta w_j \leftarrow \mathcal{N}(0, \sigma_d)$;
      $W \leftarrow argmin(\mathcal{F}(x, l, W + \Delta W_j),$
        $\mathcal{F}(x, l, W), \mathcal{F}(x, l, W - \Delta W_j))$
    end
    $d \leftarrow d - 1$
  end
end
return W

First, the weights are initialized by following the Gaussian distribution $N(0, \sqrt{2/|w_{i_d}|})$ assuming that the initial weights of convolutional neural network is already close to the final solution. $|w_{i_d}|$ means the number of parameters in the weight set w_{i_d}. Then compute standard deviation of all elements in the weight tensor W_d.

Next, weight update is performed. The weights of the layer closest to the labels are updated first and then sequentially move closer to the input. For each weight, the change is randomly sampled from Gaussian distribution $N(0, \sigma_d)$, in which the standard deviation computed in the initialization step. Then losses are computed for three different weight change scenario $(W + \Delta W_j, W, W - \Delta W_j)$ and compared. The weight set that gives minimum loss value is taken.

To note again, this update is perfomed on one weight at a time for every weights. Through C number of rounds (epochs) of these updates, model can be improved further.

1-3. RSO function

```
RSO (generic function with 1 method)
```

```
function RSO(train_loader, test_loader, C,model, batch_size, device, args)
    """
    model = convolutional model structure
    C = Number of rounds to update parameters (epochs)
    batch_size = size of the mini batch that will be used to calculate loss
    device = CPU or GPU
    """

    # Evaluate initial weight
    test_loss, test_acc = loss_and_accuracy(test_loader, model, device)
    println("Initial Weight:")
    println("   test_loss = $test_loss, test_accuracy = $test_acc")

    random_batch = []
    for (x, l) in train_loader
        push!(random_batch, (x,l))
    end

    # Initialize weights
    std_prep = []
    σ_d = Float64[]
    D = 0
    for layer in model
        D += 1
        Wd = Flux.params(layer)
        # Initialize the weights of the network with Gaussian distribution
        for id in Wd
            if typeof(id) == Array{Float32, 4}
                wj = convert(Array{Float32, 4}, rand(Normal(0, sqrt(2/length(id))),
size(id)))
            elseif typeof(id) == Vector{Float32}
                wj = convert(Vector{Float32}, rand(Normal(0, sqrt(2/length(id))),
length(id)))
            elseif typeof(id) == Matrix{Float32}
                wj = convert(Matrix{Float32}, rand(Normal(0, sqrt(2/length(id))),
size(id)))
            end
            id = wj
            append!(std_prep, vec(wj))
        end
        # Compute std of all elements in the weight tensor Wd
        push!(σ_d, std(std_prep))
    end

    # Weight update
    for c in 1:C
        d = D
        # First update the weights of the layer closest to the labels
        # and then sequentially move closer to the input
        while d > 0
            Wd = Flux.params(model[d])
            for id in Wd
                # Randomly sample change in weights from Gaussian distribution
                for j in 1:length(id)
                    # Randomly sample mini-batch
                    (x, y) = rand(random_batch, 1)[1]
                    x, y = device(x), device(y)
```

```
                # Sample a weight from normal distribution
                ΔWj = rand(Normal(0, σ_d[d]), 1)[1]

                # Weight update with three scenario
                ## F(x,l, W+ΔWj)
```

2. Experiment - compare RSO & SGD

Here we will use MNIST dataset for the training. MNIST dataset is consists of 60,000 training images and 10,000 test images of handwritten digits. Each image is a 28x28 pixel gray-scale image. Based on this dataset, we will compare the classification accuracy of RSO and SGD. The box below is the training variables that can be changed.

Args

```
@kwdef mutable struct Args
    tracker::Tracker=notrack    # track loss or accuracy
    use_cuda::Bool = false      # use gpu (if gpu available - not tested)
    η::Float64 = 0.01           # learning rate
    batchsize::Int = 256        # batch size
    epochs::Int = 1             # number of epochs
    dataset::String = "MNIST"   # MNIST or CIFAR10
    optimiser::String = "SGD"   # SGD or RSO
    model::String = "OneConv"   # Original or TwoConv or OneConv
end
```

2-1. SGD (Backpropagation)

For OneConv_model with 32 batch size, it takes around 100 seconds to compute 10 epochs.

```
begin
    acc_tracker_SGD = TrackObj(Float32)
    train(epochs=10, tracker=acc_tracker_SGD, batchsize=32)
end
```

2-2. RSO

For OneConv_model with 32 batch size, it takes around 200 seconds to compute 10 epochs.

```
begin
    acc_tracker_RSO = TrackObj(Float32)
    train(epochs=10, optimiser="RSO", tracker=acc_tracker_RSO, batchsize=32)
end
```

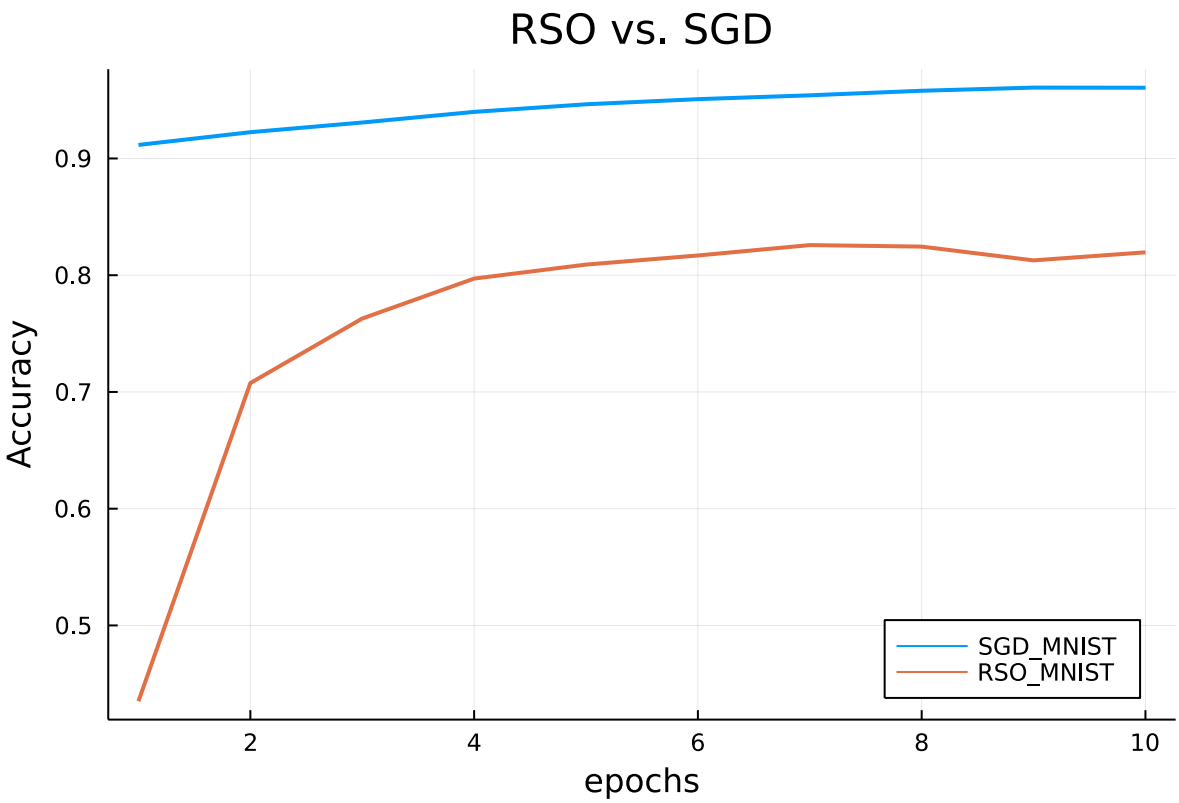
2-3. Result

1) RSO performance summary:

	Round	batch	time(s)	loss	acc
Original model	1	256	5240	-	-
	1	1000	73042	-	-
TwoConv model	10	32	1457	0.5126024	0.8504
	10	512	19186	0.19334497	0.9397
	10	1000	36464	0.12634973	0.9604
OneConv model	10	32	212	0.6316686	0.8075
	10	64	345	0.55254054	0.8353
	10	256	1641	0.37202972	0.8929
	10	512	2834	0.29727805	0.9152
	10	1024	5046	0.3011675	0.9148

This table is generated by myself through several experiments to show the performance of RSO in different conditions. Please note that these experiments could not be generated user-interactively since the running time was insanely long for certain experiments. Different models are explained in `3-1.2) Model Structures`. The loss and accuracy are not recorded in original model since it took immensive amount of time in training with original model even for 1 round (epoch). In both simpler models, it can be observed that the loss and accuracy improves as the batch size increases. This is because the loss calculation for weight updates were based on larger samples which lead to more precise reflection of the model (line 63, 68, 73 in RSO function). Furthermore, the generall loss and accuracy was better in TwoConv model than OneConv model.

2) Accuracy Comparison



Although it seems that RSO is lagging behind SGD in this plot, we have to consider that it was not the best model that can be applied due to practical reasons. As shown above in `1) RSO performance summary`, the accuracy can peak 96.04% even with a little bit of more complexity in the model with larger batch size. According to the paper, RSO can perform 99.12% of accuracy after 50 cycles of updates with MNIST dataset, while the SGD gives 99.27% of accuracy after 50 epochs. Thus, one may consider using random sampling method than back-propagation methods for training neural networks as proposed and demonstrated in the paper.

3. Appendix (Source code)

3-1. CNN training functions

The baseline of the training structure is referred from FluxML/model-zoo tutorial notebook [2].

1) Create mini-batch iterators (DataLoaders)

Prepare dataset for training. This involves loading data, adding channel layer, encoding, and creating DataLoader object.

```
getdata (generic function with 1 method)
```

```
function getdata(args, device)
    ENV["DATADEPS_ALWAYS_ACCEPT"] = "true"

    # load data
    if args.dataset == "MNIST"
        x_train, y_train = MNIST.traindata(Float32)
        x_test, y_test = MNIST.testdata(Float32)
    elseif args.dataset == "CIFAR10"
        x_train, y_train = CIFAR10.traindata(Float32)
        x_test, y_test = CIFAR10.testdata(Float32)
    end

    # Add channel layer
    # The unsqueeze() function helps image data to be in order of (width, height, #channels, batch size)
    x_train = Flux.unsqueeze(x_train, 3)
    x_test = Flux.unsqueeze(x_test, 3)

    # Encode labels
    y_train = onehotbatch(y_train, 0:9)
    y_test = onehotbatch(y_test, 0:9)

    # Create DataLoaders (mini-batch iterators)
    train_loader = DataLoader((x_train, y_train), batchsize=args.batchsize, shuffle=true)
    test_loader = DataLoader((x_test, y_test), batchsize=args.batchsize)

    return train_loader, test_loader
end
```

2) Model structures

- **Original model:** Original model is the model described in the paper. It is comprised of 6 convolutional layers with mean pooling layer in every two convolutional layers.
- **TwoConv model:** Simplified model for algorithm test. It contains two convolutional layers.
- **OneConv model:** Further simplified model for algorithm test. It contains only one convolutional layer.

```
original_model (generic function with 1 method)
```

```
function original_model(; imgsize=(28, 28, 1), nclasses=10)
    # This is the model described in the paper
    return Chain(
    # input 28x28x1
    Conv((3,3), 1=>16, pad=1), BatchNorm(16, relu),      # 28x28x16
    Conv((3,3), 16=>16, pad=1), BatchNorm(16, relu),     # 28x28x16
    MeanPool((2, 2)),                                    # 14x14x16
    Conv((3,3), 16=>16, pad=1), BatchNorm(16, relu),     # 14x14x16
    Conv((3,3), 16=>16, pad=1), BatchNorm(16, relu),     # 14x14x16
    MeanPool((2, 2)),                                    # 7x7x16
    Conv((3,3), 16=>16, pad=1), BatchNorm(16, relu),     # 7x7x16
    Conv((3,3), 16=>16, pad=1), BatchNorm(16, relu),     # 7x7x16

    # Average pooling on each width x height feature map
    GlobalMeanPool(),
    # Remove 1x1 dimensions (singletons)
    flatten,

    Dense(16, nclasses),
    softmax)
end
```

```
TwoConv_model (generic function with 1 method)
```

```
function TwoConv_model(; imgsize=(28, 28, 1), nclasses=10)
    # Simpler model to test algorithm
    cnn_output_size = Int.(floor.([imgsize[1]/4,imgsize[2]/4,16]))

    return Chain(
    # input 28x28x1
    Conv((3,3), 1=>16, pad=1, relu),    #14x14x16
    MaxPool((2,2)),
    Conv((3,3), 16=>16, pad=1, relu),   #7x7x16
    MaxPool((2,2)),

    flatten,

    Dense(prod(cnn_output_size), nclasses))
end
```

```
OneConv_model (generic function with 1 method)
```

```
function OneConv_model(; imgsize=(28, 28, 1), nclasses=10)
    # Simpler model to test algorithm
    cnn_output_size = Int.(floor.([imgsize[1]/2,imgsize[2]/2,4]))

    return Chain(
    # input 28x28x1
    Conv((3,3), 1=>4, pad=1, relu),     #14x14x16
    MaxPool((2,2)),

    flatten,

    Dense(prod(cnn_output_size), nclasses))
end
```

3) Loss and accuracy

```
loss_and_accuracy (generic function with 1 method)
```

```
function loss_and_accuracy(data_loader, model, device)
    acc = 0
    ls = 0.0f0
    num = 0
    for (x, y) in data_loader
        x, y = device(x), device(y)
        ŷ = model(x)
        ls += logitcrossentropy(ŷ, y, agg=sum)
        acc += sum(onecold(ŷ) .== onecold(y))
        num += size(x)[end]
    end
    return ls/num, acc/num
end
```

4) Training main body

```
train (generic function with 1 method)
```

```
function train(; kws...)
    args = Args(; kws...) # collect options in a stuct for convinience

    # Choose device
    ## WARNING: GPU not tested
    if CUDA.functional() && args.use_cuda
        @info "Training on CUDA GPU"
        CUDA.allowscalar(false)
        device = gpu
    else
        @info "Training on CPU"
        device = cpu
    end

    # Prepare datasets
    train_loader, test_loader = getdata(args, device)

    # Construct model
    if args.model == "original"
        model = original_model() |> device
    elseif args.model == "TwoConv"
        model = TwoConv_model() |> device
    elseif args.model == "OneConv"
        model = OneConv_model() |> device
    end

    ps = Flux.params(model) # model's trainable parameters

    best_param = ps

    ## Training
    if args.optimiser == "SGD"
        ## Optimizer
        opt = Descent(args.η)

        best_loss = 10
        best_acc = 0
        for epoch in 1:args.epochs
            for (x, y) in train_loader
                x, y = device(x), device(y) # transfer data to device
                # compute gradient
                gs = gradient(() -> logitcrossentropy(model(x), y), ps)
                Flux.Optimise.update!(opt, ps, gs) # update parameters
            end

            # Report on train and test
            train_loss, train_acc = loss_and_accuracy(train_loader, model, device)
            test_loss, test_acc = loss_and_accuracy(test_loader, model, device)

            # track accuracy of the model
            track!(args.tracker, test_acc)

            # Save the best model
            if test_loss < best_loss
                best_param = ps
                best_loss = test_loss
                best_acc = test_acc
```

```
            end

            println("Epoch=$epoch")
            println("   train_loss = $train_loss, train_accuracy = $train_acc")
            println("   test_loss = $test_loss, test_accuracy = $test_acc")
        end


    elseif args.optimiser == "RSO"
        # Run RSO function and update ps
        best_param = RSO(train_loader, test_loader, args.epochs, model,
                         args.batchsize, device, args)
        best_loss, best_acc = loss_and_accuracy(test_loader, model, device)
    end


    println("Best test loss = $best_loss, Best test accuracy = $best_acc")
end
```

3-2. Tracker

These tracker functions are brought directly from the lecture notebook `searching_methods.jl`. A tracker is a data structure to keep track of the training loss during the run of the algorithm.

```
abstract type Tracker end
```

```
struct NoTracking <: Tracker end
```

```
notrack = ▸NoTracking()
```

```
notrack = NoTracking()
```

```
struct TrackObj{T} <: Tracker
    objectives::Vector{T}
    TrackObj(T::Type=Float32) = new{T}([])
end
```

```
track! (generic function with 1 method)
```

```
track!(::NoTracking, loss_acc) = nothing
```

```
track! (generic function with 2 methods)
```

```
track!(tracker::TrackObj, loss_acc) = push!(tracker.objectives, loss_acc)
```

4. Unit test

```
▸DefaultTestSet("Unit test", [], 3, false, false)
```

References

[1] Tripathi, R., & Singh, B. (2020). RSO: A Gradient Free Sampling Based Approach For Training Deep Neural Networks. arXiv preprint arXiv:2005.05955. **Link to paper**

[2] **FluxML/model-zoo/vision/mlp_mnist/mlp_mnist.jl**

[3] J. Frankle, D. J. Schwab, and A. S. Morcos. Training batchnorm and only batchnorm: On the expressive power of random features in cnns. arXiv preprint arXiv:2003.00152, 2020.