Overview

Most optimization algorithms have the following interface:

```
x*, {f}, ... = optim.method(opfunc, x[, config][, state])
```

where:

- `opfunc` : a user-defined closure that respects this API: `f, df/dx = func(x)`
- `x` : the current parameter vector (a 1D `Tensor`)
- `config` : a table of parameters, dependent upon the algorithm
- `state` : a table of state variables, if `nil` , `config` will contain the state
- `x*` : the new parameter vector that minimizes `f, x* = argmin_x f(x)`
- `{f}` : a table of all `f` values, in the order they've been evaluated (for some simple algorithms, like SGD, `#f == 1`)

Example

The state table is used to hold the state of the algorithm.
It's usually initialized once, by the user, and then passed to the optim function as a black box.
Example:

```
config = {
   learningRate = 1e-3,
   momentum = 0.5
}

for i, sample in ipairs(training_samples) do
    local func = function(x)
       -- define eval function
       return f, df_dx
    end
    optim.sgd(func, x, config)
end
```

Training using optim

`optim` is a quite general optimizer, for minimizing any function with respect to a set of parameters.
In our case, our function will be the loss of our network, given an input, and a set of weights.
The goal of training a neural net is to optimize the weights to give the lowest loss over our validation set, by using the training set as a proxy.
So, we are going to use optim to minimize the loss with respect to the weights, over our training set.

To illustrate all the steps required, we will go over a simple example, where we will train a neural network on the classical XOR problem.
We will feed the data to `optim` in minibatches (we will use here just one minibatch), breaking your training set into chucks, and feed each minibatch to `optim`, one by one.

We need to give `optim` a function that will output the loss and the derivative of the loss with respect to the
weights, given the current weights, as a function parameter.
The function will have access to our training minibatch, and use this to calculate the loss, for this minibatch.
Typically, the function would be defined inside our loop over batches, and therefore have access to the current minibatch data.

Neural Network

We create a simple neural network with one hidden layer.

```
require 'nn'

model = nn.Sequential()  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20 -- parameters
model:add(nn.Linear(inputs, HUs))
model:add(nn.Tanh())
model:add(nn.Linear(HUs, outputs))
```

> If we would like to train on GPU, then we need to shipt the model to *device memory* by typing `model:cuda()` after having issued `require 'cunn'`.

Criterion

We choose the *Mean Squared Error* loss `Criterion` :

```
criterion = nn.MSECriterion()
```

We are using an `nn.MSECriterion` because we are training on a regression task, predicting contiguous (real) target value, from `-1` to `+1` .
For a classification task, with more than two classes, we would add an `nn.LogSoftMax` layer to the end of our network, and use a `nn.ClassNLLCriterion` loss criterion.
Nevertheless, the XOR problem could be seen and a two classes classification task, associated to the `-1` and `+1` discrete outputs.

> If we would like to train on GPU, then we need to ship the `Criterion` to *device memory* by typing `criterion:cuda()` .

Data set

We will just create one minibatch of `128` examples.
In your own training, you'd want to break down your rather larger data set into multiple minibatches, of around `32` to `512` examples each.

```
batchSize = 128
batchInputs = torch.DoubleTensor(batchSize, inputs) -- or
CudaTensor for GPU training
batchLabels = torch.DoubleTensor(batchSize)         -- or
CudaTensor for GPU training

for i = 1, batchSize do
   local input = torch.randn(2)     -- normally distributed example
in 2d
   local label
   if input[1] * input[2] > 0 then  -- calculate label for XOR
function
      label = -1
   else
      label = 1
   end
```

```
      batchInputs[i]:copy(input)
      batchLabels[i] = label
   end
```

Flatten parameters

`optim` expects the parameters that are to be optimized, and their gradients, to be one-dimensional `Tensor`s.
But, our network model contains probably multiple modules, typically multiple convolutional layers, and each of these layers has their own `weight` and `bias` `Tensor`s.
How to handle this?

It is simple: we can call a standard method `:getParameters()`, that is defined for any network module.
When we call this method, the following magic will happen:

- a new `Tensor` will be created, large enough to hold all the `weight`s and `bias`es of the entire network model
- the model `weight` and `bias` `Tensor`s are replaced with views onto the new contiguous parameter `Tensor`
- and the exact same thing will happen for all the gradient `Tensor`s: replaced with views onto one single contiguous gradient `Tensor`

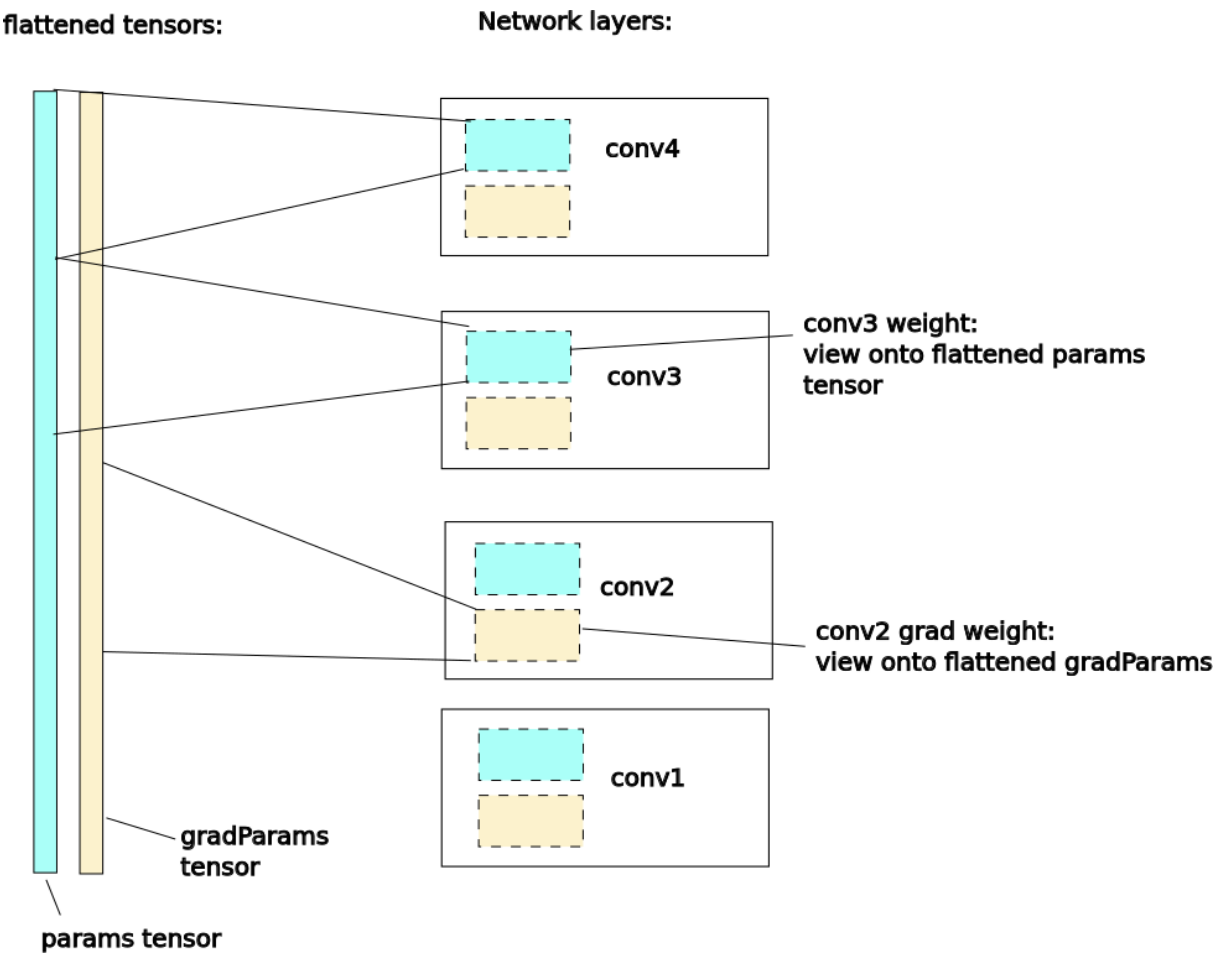
We can call this method as follows:

```
params, gradParams = model:getParameters()
```

These flattened `Tensor`s have the following characteristics:

- to `optim`, the parameters it needs to optimize are all contained in one single one-dimensional `Tensor`
- when `optim` optimizes the parameters in this large one-dimensional `Tensor`, it is implicitly optimizing the `weight`s and `bias`es in our network model, since those are now simply views onto this large one-dimensional parameter `Tensor`

It will look something like this:



> Note that flattening the parameters redefines the `weight` and `bias` `Tensor`s for all the network modules in our network model.
> Therefore, any pre-existing references to the original model layer `weight` and `bias` `Tensor`s will no longer point to the model `weight` and `bias` `Tensor`s, after flattening.

Training

Now that we have created our model, our training set, and prepared the flattened network parameters, we can train using `optim`.
`optim` provides various training algorithms.
We will use the stochastic gradient descent algorithm SGD.
We need to provide the learning rate, via an optimization state table:

```
local optimState = {learningRate = 0.01}
```

We define an evaluation function, inside our training loop, and use `optim.sgd` to train the system:

```
require 'optim'

for epoch = 1, 50 do
   -- local function we give to optim
   -- it takes current weights as input, and outputs the loss
   -- and the gradient of the loss with respect to the weights
   -- gradParams is calculated implicitly by calling 'backward',
   -- because the model's weight and bias gradient tensors
   -- are simply views onto gradParams
   function feval(params)
      gradParams:zero()

      local outputs = model:forward(batchInputs)
      local loss = criterion:forward(outputs, batchLabels)
      local dloss_doutputs = criterion:backward(outputs,
batchLabels)
      model:backward(batchInputs, dloss_doutputs)

      return loss, gradParams
   end
   optim.sgd(feval, params, optimState)
end
```

Test the network

For the prediction task, we will also typically use minibatches, although we can run prediction sample by sample too.
In this example, we will predict sample by sample.
To run prediction on a minibatch, simply pass in a tensor with one additional dimension, which represents the sample index.

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(model:forward(x))
x[1] =  0.5; x[2] = -0.5; print(model:forward(x))
x[1] = -0.5; x[2] =  0.5; print(model:forward(x))
x[1] = -0.5; x[2] = -0.5; print(model:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(model:forward(x))

-0.3490
[torch.DoubleTensor of dimension 1]

> x[1] =  0.5; x[2] = -0.5; print(model:forward(x))

 1.0561
[torch.DoubleTensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(model:forward(x))

 0.8640
[torch.DoubleTensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(model:forward(x))

-0.2941
[torch.DoubleTensor of dimension 1]
```

If we were running on a GPU, we would probably want to predict using minibatches, because this will hide the latencies involved in transferring data from main memory to the GPU.
To predict on a minbatch, we could do something like:

```
x = torch.CudaTensor({
   { 0.5,  0.5},
   { 0.5, -0.5},
   {-0.5,  0.5},
   {-0.5, -0.5}
})
print(model:forward(x))
```

You should see something like:

```
> print(model:forward(x))
 -0.3490
  1.0561
  0.8640
 -0.2941
```

```
 [torch.CudaTensor of size 4]
```

That's it!
For minibatched prediction, the output tensor contains one value for each of our input data samples.