Overview

Each module of a network is composed of Modules and there are several sub-classes of `Module` available: container classes like Sequential, Parallel and Concat , which can contain simple layers like Linear, Mean, Max and Reshape, as well as convolutional layers, and transfer functions like Tanh.

Loss functions are implemented as sub-classes of Criterion. They are helpful to train neural network on classical tasks. Common criterions are the Mean Squared Error criterion implemented in MSECriterion and the cross-entropy criterion implemented in ClassNLLCriterion.

Finally, the StochasticGradient class provides a high level way to train the neural network of choice, even though it is easy with a simple for loop to train a neural network yourself.

Detailed Overview

This section provides a detailed overview of the neural network package. First the omnipresent Module is examined, followed by some examples for combining modules together. The last part explores facilities for training a neural network, and finally some caveats while training networks with shared parameters.

Module

A neural network is called a Module (or simply *module* in this documentation) in Torch. `Module` is an abstract class which defines four main methods:

- forward(input) which computes the output of the module given the `input` Tensor.
- backward(input, gradOutput) which computes the gradients of the module with respect

to its own parameters, and its own inputs.
- zeroGradParameters() which zeroes the gradient with respect to the parameters of the module.
- updateParameters(learningRate) which updates the parameters after one has computed the gradients with `backward()`

It also declares two members:

- output which is the output returned by `forward()`.
- gradInput which contains the gradients with respect to the input of the module, computed in a `backward()`.

Two other perhaps less used but handy methods are also defined:

- share(mlp,s1,s2,…,sn) which makes this module share the parameters s1,..sn of the module `mlp`. This is useful if you want to have modules that share the same weights.
- clone(…) which produces a deep copy of (i.e. not just a pointer to) this Module, including the current state of its parameters (if any).

Some important remarks:

- `output` contains only valid values after a forward(input).
- `gradInput` contains only valid values after a backward(input, gradOutput).
- backward(input, gradOutput) uses certain computations obtained during forward(input). You *must* call `forward()` before calling a `backward()`, on the *same* `input`, or your gradients are going to be incorrect!

Plug and play

Building a simple neural network can be achieved by constructing an available layer.
A linear neural network (perceptron!) is built only in one line:

```
mlp = nn.Linear(10,1) -- perceptron with 10 inputs
```

More complex neural networks are easily built using container classes Sequential and Concat. `Sequential` plugs layer in a feed-forward fully connected manner. `Concat` concatenates in one layer several modules: they take the same inputs, and their output is concatenated.

Creating a one hidden-layer multi-layer perceptron is thus just as easy as:

```
mlp = nn.Sequential()
mlp:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
mlp:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
mlp:add( nn.Linear(25, 1) ) -- 1 output
```

Of course, `Sequential` and `Concat` can contains other `Sequential` or `Concat`, allowing you to try the craziest neural networks you ever dreamt of!

Training a neural network

Once you built your neural network, you have to choose a particular Criterion to train it. A criterion is a class which describes the cost to be minimized during training.

You can then train the neural network by using the StochasticGradient class.

```
 criterion = nn.MSECriterion() -- Mean Squared Error criterion
 trainer = nn.StochasticGradient(mlp, criterion)
 trainer:train(dataset) -- train using some examples
```

StochasticGradient expect as a `dataset` an object which implements the operator `dataset[index]` and implements the method `dataset:size()`. The `size()` methods returns the number of examples and `dataset[i]` has to return the i-th example.

An `example` has to be an object which implements the operator `example[field]`, where `field` might take the value `1` (input features) or `2` (corresponding label which will be given to the criterion). The input is usually a Tensor (except if you use special kind of gradient modules, like table layers). The label type depends on the criterion. For example, the MSECriterion expect a Tensor, but the ClassNLLCriterion except a integer number (the class).

Such a dataset is easily constructed by using Lua tables, but it could any `C` object for example, as long as required operators/methods are implemented. See an example.

`StochasticGradient` being written in `Lua`, it is extremely easy to cut-and-paste it and create a variant to it adapted to your needs (if the constraints of `StochasticGradient` do not satisfy you).

Low Level Training

If you want to program the `StochasticGradient` by hand, you essentially need to control the use of forwards and backwards through the network yourself. For example, here is the code fragment one would need to make a gradient step given an input `x`, a desired output `y`, a network `mlp` and a given criterion `criterion` and learning rate `learningRate`:

```lua
function gradUpdate(mlp, x, y, criterion, learningRate)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  mlp:updateParameters(learningRate)
end
```

For example, if you wish to use your own criterion you can simply replace `gradCriterion` with the gradient vector of your criterion of choice.

A Note on Sharing Parameters

By using `:share(...)` and the Container Modules, one can easily create very complex architectures. In order to make sure that the network is going to train properly, one needs to pay attention to the way the sharing is applied, because it might depend on the optimization procedure.

- If you are using an optimization algorithm that iterates over the modules of your network (by calling `:updateParameters` for example), only the parameters of the network should be shared.
- If you use the flattened parameter tensor to optimize the network, obtained by calling `:getParameters`, for example for the package `optim`, then you need to share both the parameters and the gradParameters.

Here is an example for the first case:

```
-- our optimization procedure will iterate over the modules, so
only share
-- the parameters
mlp = nn.Sequential()
linear = nn.Linear(2,2)
linear_clone = linear:clone('weight','bias') -- clone sharing the
parameters
mlp:add(linear)
mlp:add(linear_clone)
function gradUpdate(mlp, x, y, criterion, learningRate)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  mlp:updateParameters(learningRate)
end
```

And for the second case:

```
-- our optimization procedure will use all the parameters at once,
because
-- it requires the flattened parameters and gradParameters Tensors.
Thus,
-- we need to share both the parameters and the gradParameters
mlp = nn.Sequential()
linear = nn.Linear(2,2)
-- need to share the parameters and the gradParameters as well
linear_clone =
linear:clone('weight','bias','gradWeight','gradBias')
mlp:add(linear)
mlp:add(linear_clone)
params, gradParams = mlp:getParameters()
function gradUpdate(mlp, x, y, criterion, learningRate, params,
gradParams)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  -- adds the gradients to all the parameters at once
  params:add(-learningRate, gradParams)
end
```