Neural Network Graph Package

This package provides graphical computation for `nn` library in Torch.

Requirements

You do *not* need `graphviz` to be able to use this library, but if you have it you will be able to display the graphs that you have created. For installing the package run the appropriate command below:

```
# Mac users
brew install graphviz
# Debian/Ubuntu users
sudo apt-get install graphviz -y
```

Usage

Plug: A more explanatory nngraph tutorial by Nando De Freitas of Oxford

The aim of this library is to provide users of `nn` package with tools to easily create complicated architectures.
Any given `nn` `module` is going to be bundled into a *graph node*.
The `__call__` operator of an instance of `nn.Module` is used to create architectures as if one is writing function calls.

Two hidden layers MLP

```
h1 = nn.Linear(20, 10)()
h2 = nn.Linear(10, 1)(nn.Tanh()(nn.Linear(10, 10)(nn.Tanh()(h1))))
mlp = nn.gModule({h1}, {h2})
```

```
x = torch.rand(20)
dx = torch.rand(1)
mlp:updateOutput(x)
mlp:updateGradInput(x, dx)
mlp:accGradParameters(x, dx)

-- draw graph (the forward graph, '.fg')
graph.dot(mlp.fg, 'MLP')
```

MLP

Read this diagram from top to bottom, with the first and last nodes being *dummy nodes* that regroup all inputs and outputs of the graph.
The `module` entry describes the function of the node, as applies to `input`, and producing a result of the shape `gradOutput`; `mapindex` contains pointers to the parent nodes.

To save the *graph* on file, specify the file name, and both a `dot` and `svg` files will be saved. For example, you can type:

```
graph.dot(mlp.fg, 'MLP', 'myMLP')
```

You can also use the `__unm__` and `__sub__` operators to replace all `__call__`:

```
h1 = - nn.Linear(20,10)
h2 = h1
     - nn.Tanh()
     - nn.Linear(10,10)
     - nn.Tanh()
     - nn.Linear(10, 1)
mlp = nn.gModule({h1}, {h2})
```

A network with 2 inputs and 2 outputs

```
h1 = nn.Linear(20, 20)()
h2 = nn.Linear(10, 10)()
hh1 = nn.Linear(20, 1)(nn.Tanh()(h1))
hh2 = nn.Linear(10, 1)(nn.Tanh()(h2))
```

```
madd = nn.CAddTable()({hh1, hh2})
oA = nn.Sigmoid()(madd)
oB = nn.Tanh()(madd)
gmod = nn.gModule({h1, h2}, {oA, oB})

x1 = torch.rand(20)
x2 = torch.rand(10)

gmod:updateOutput({x1, x2})
gmod:updateGradInput({x1, x2}, {torch.rand(1), torch.rand(1)})
graph.dot(gmod.fg, 'Big MLP')
```

Alternatively, you can use `-` to make your code looks like the data flow:

```
h1 = - nn.Linear(20,20)
h2 = - nn.Linear(10,10)
hh1 = h1 - nn.Tanh() - nn.Linear(20,1)
hh2 = h2 - nn.Tanh() - nn.Linear(10,1)
madd = {hh1,hh2} - nn.CAddTable()
oA = madd - nn.Sigmoid()
oB = madd - nn.Tanh()
gmod = nn.gModule( {h1,h2}, {oA,oB} )
```

Big MLP

A network with containers

Another net that uses container modules (like `ParallelTable`) that output a table of outputs.

```
m = nn.Sequential()
m:add(nn.SplitTable(1))
m:add(nn.ParallelTable():add(nn.Linear(10, 20)):add(nn.Linear(10,
30)))
input = nn.Identity()()
input1, input2 = m(input):split(2)
m3 = nn.JoinTable(1)({input1, input2})

g = nn.gModule({input}, {m3})

indata = torch.rand(2, 10)
```

```
gdata = torch.rand(50)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph')
graph.dot(g.bg, 'Backward Graph')
```

Forward Graph

Backward Graph

More fun with graphs

A multi-layer network where each layer takes output of previous two layers as input.

```
input = nn.Identity()()
L1 = nn.Tanh()(nn.Linear(10, 20)(input))
L2 = nn.Tanh()(nn.Linear(30, 60)(nn.JoinTable(1)({input, L1})))
L3 = nn.Tanh()(nn.Linear(80, 160)(nn.JoinTable(1)({L1, L2})))

g = nn.gModule({input}, {L3})

indata = torch.rand(10)
gdata = torch.rand(160)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph')
graph.dot(g.bg, 'Backward Graph')
```

As your graph getting bigger and more complicated, the nested parentheses may become confusing. In this case, using `-` to chain the modules is a clearer and easier way:

```
input = - nn.Identity()
L1 =  input
     - nn.Linear(10, 20)
     - nn.Tanh()
L2 =  { input, L1 }
     -  nn.JoinTable(1)
     -  nn.Linear(30,60)
```

```
      -  nn.Tanh()
L3 = { L1,L2 }
      - nn.JoinTable(1)
      - nn.Linear(80,160)
      - nn.Tanh()
g = nn.gModule({input},{L3})
```

Forward Graph

Backward Graph

Annotations

It is possible to add annotations to your network, such as labeling nodes with names or attributes which will show up when you graph the network.
This can be helpful in large graphs.

For the full list of graph attributes see the
graphviz documentation.

```
input = nn.Identity()()
L1 = nn.Tanh()(nn.Linear(10, 20)(input)):annotate{
   name = 'L1', description = 'Level 1 Node',
   graphAttributes = {color = 'red'}
}
L2 = nn.Tanh()(nn.Linear(30, 60)(nn.JoinTable(1)({input,
L1}))):annotate{
   name = 'L2', description = 'Level 2 Node',
   graphAttributes = {color = 'blue', fontcolor = 'green'}
}
L3 = nn.Tanh()(nn.Linear(80, 160)(nn.JoinTable(1)({L1,
L2}))):annotate{
   name = 'L3', description = 'Level 3 Node',
   graphAttributes = {color = 'green',
   style = 'filled', fillcolor = 'yellow'}
}

g = nn.gModule({input},{L3})
```

```
indata = torch.rand(10)
gdata = torch.rand(160)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph', '/tmp/fg')
graph.dot(g.bg, 'Backward Graph', '/tmp/bg')
```

In this case, the graphs are saved in the following 4 files: `/tmp/{fg,bg}.{dot,svg}`.

Forward Graph

Backward Graph

Debugging

With nngraph, one can create very complicated networks. In these cases, finding errors can be hard. For that purpose, nngraph provides several useful utilities. The following code snippet shows how to use local variable names for annotating the nodes in a graph and how to enable debugging mode that automatically creates an svg file with error node marked in case of a runtime error.

```
require 'nngraph'

-- generate SVG of the graph with the problem node highlighted
-- and hover over the nodes in svg to see the filename:line_number
info
-- nodes will be annotated with local variable names even if debug
mode is not enabled.
nngraph.setDebug(true)

local function get_net(from, to)
    local from = from or 10
    local to = to or 10
    local input_x = nn.Identity()()
    local linear_module = nn.Linear(from, to)(input_x)

    -- Annotate nodes with local variable names
```

```
    nngraph.annotateNodes()
    return nn.gModule({input_x},{linear_module})
end

local net = get_net(10,10)

-- if you give a name to the net, it will use that name to produce
the
-- svg in case of error, if not, it will come up with a name
-- that is derived from number of inputs and outputs to the graph
net.name = 'my_bad_linear_net'

-- prepare an input that is of the wrong size to force an error
local input = torch.rand(11)
pcall(function() net:updateOutput(input) end)
-- it should have produced an error and spit out a graph
-- just run Safari to display the svg
os.execute('open -a  Safari my_bad_linear_net.svg')
```