Containers

Complex neural networks are easily built using container classes:

- Container : abstract class inherited by containers ;
  - Sequential : plugs layers in a feed-forward fully connected manner ;
  - Parallel : applies its `ith` child module to the `ith` slice of the input Tensor ;
  - Concat : concatenates in one layer several modules along dimension `dim` ;
    - DepthConcat : like Concat, but adds zero-padding when non-`dim` sizes don't match;
  - Bottle : allows any dimensionality input be forwarded through a module ;

See also the Table Containers for manipulating tables of Tensors.

Container

This is an abstract Module class which declares methods defined in all containers.
It reimplements many of the Module methods such that calls are propagated to the contained modules. For example, a call to zeroGradParameters will be propagated to all contained modules.

add(module)

Adds the given `module` to the container. The order is important

get(index)

Returns the contained modules at index `index`.

size()

Returns the number of contained modules.

Sequential

Sequential provides a means to plug layers together
in a feed-forward fully connected manner.

E.g.
creating a one hidden-layer multi-layer perceptron is thus just as easy as:

```
mlp = nn.Sequential()
mlp:add(nn.Linear(10, 25)) -- Linear module (10 inputs, 25 hidden
units)
mlp:add(nn.Tanh())         -- apply hyperbolic tangent transfer
function on each hidden units
mlp:add(nn.Linear(25, 1))  -- Linear module (25 inputs, 1 output)

> mlp
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.Linear(10 -> 25)
  (2): nn.Tanh
  (3): nn.Linear(25 -> 1)
}

> print(mlp:forward(torch.randn(10)))
-0.1815
[torch.Tensor of dimension 1]
```

remove([index])

Remove the module at the given `index`. If `index` is not specified, remove the last layer.

```
model = nn.Sequential()
model:add(nn.Linear(10, 20))
model:add(nn.Linear(20, 20))
model:add(nn.Linear(20, 30))
```

```
model:remove(2)
> model
nn.Sequential {
  [input -> (1) -> (2) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Linear(20 -> 30)
}
```

insert(module, [index])

Inserts the given `module` at the given `index`. If `index` is not specified, the incremented length of the sequence is used and so this is equivalent to use `add(module)`.

```
model = nn.Sequential()
model:add(nn.Linear(10, 20))
model:add(nn.Linear(20, 30))
model:insert(nn.Linear(20, 20), 2)
> model
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Linear(20 -> 20)      -- The inserted layer
  (3): nn.Linear(20 -> 30)
}
```

Parallel

`module` = `Parallel(inputDimension,outputDimension)`

Creates a container module that applies its `ith` child module to the `ith` slice of the input Tensor by using select
on dimension `inputDimension`. It concatenates the results of its contained modules together along dimension `outputDimension`.

Example:

```
mlp = nn.Parallel(2,1);   -- Parallel container will associate a
```

```
module to each slice of dimension 2
                           -- (column space), and concatenate the
outputs over the 1st dimension.

mlp:add(nn.Linear(10,3)); -- Linear module (input 10, output 3),
applied on 1st slice of dimension 2
mlp:add(nn.Linear(10,2))  -- Linear module (input 10, output 2),
applied on 2nd slice of dimension 2

                                  -- After going through the Linear
module the outputs are
                                  -- concatenated along the unique
dimension, to form 1D Tensor
> mlp:forward(torch.randn(10,2)) -- of size 5.
-0.5300
-1.1015
 0.7764
 0.2819
-0.6026
[torch.Tensor of dimension 5]
```

A more complicated example:

```

mlp = nn.Sequential();
c = nn.Parallel(1,2)     -- Parallel container will associate a
module to each slice of dimension 1
                         -- (row space), and concatenate the
outputs over the 2nd dimension.

for i=1,10 do            -- Add 10 Linear+Reshape modules in
parallel (input = 3, output = 2x1)
 local t=nn.Sequential()
 t:add(nn.Linear(3,2))   -- Linear module (input = 3, output = 2)
 t:add(nn.Reshape(2,1))  -- Reshape 1D Tensor of size 2 to 2D
Tensor of size 2x1
 c:add(t)
end

mlp:add(c)               -- Add the Parallel container in the
Sequential container

pred = mlp:forward(torch.randn(10,3)) -- 2D Tensor of size 10x3
goes through the Sequential container
```

```
                                        -- which contains a Parallel
container of 10 Linear+Reshape.
                                        -- Each Linear+Reshape module
receives a slice of dimension 1
                                        -- which corresponds to a 1D
Tensor of size 3.
                                        -- Eventually all the
Linear+Reshape modules' outputs of size 2x1
                                        -- are concatenated alond the
2nd dimension (column space)
                                        -- to form pred, a 2D Tensor
of size 2x10.

> pred
-0.7987 -0.4677 -0.1602 -0.8060  1.1337 -0.4781  0.1990  0.2665
-0.1364  0.8109
-0.2135 -0.3815  0.3964 -0.4078  0.0516 -0.5029 -0.9783 -0.5826
0.4474  0.6092
[torch.DoubleTensor of size 2x10]


for i = 1, 10000 do     -- Train for a few iterations
 x = torch.randn(10,3);
 y = torch.ones(2,10);
 pred = mlp:forward(x)

 criterion = nn.MSECriterion()
 local err = criterion:forward(pred,y)
 local gradCriterion = criterion:backward(pred,y);
 mlp:zeroGradParameters();
 mlp:backward(x, gradCriterion);
 mlp:updateParameters(0.01);
 print(err)
end
```

Concat

```
module = nn.Concat(dim)
```

Concat concatenates the output of one layer of "parallel" modules along the provided dimension `dim`: they take the same inputs, and their output is concatenated.

```
mlp = nn.Concat(1);
mlp:add(nn.Linear(5,3))
mlp:add(nn.Linear(5,7))

> print(mlp:forward(torch.randn(5)))
 0.7486
 0.1349
 0.7924
-0.0371
-0.4794
 0.3044
-0.0835
-0.7928
 0.7856
-0.1815
[torch.Tensor of dimension 10]
```

DepthConcat

```
module = nn.DepthConcat(dim)
```

DepthConcat concatenates the output of one layer of "parallel" modules along the provided dimension `dim`: they take the same inputs, and their output is concatenated. For dimensions other than `dim` having different sizes, the smaller tensors are copied in the center of the output tensor, effectively padding the borders with zeros.

The module is particularly useful for concatenating the output of Convolutions along the depth dimension (i.e. `nOutputFrame`). This is used to implement the *DepthConcat* layer of the Going deeper with convolutions article. The normal Concat Module can't be used since the spatial dimensions (height and width) of the output Tensors requiring concatenation may have different values. To deal with this, the output uses the largest spatial dimensions and adds zero-padding around the smaller Tensors.

```
inputSize = 3
outputSize = 2
input = torch.randn(inputSize,7,7)

mlp=nn.DepthConcat(1);
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 1, 1))
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 3, 3))
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 4, 4))

> print(mlp:forward(input))
(1,.,.) =
 -0.2874  0.6255  1.1122  0.4768  0.9863 -0.2201 -0.1516
  0.2779  0.9295  1.1944  0.4457  1.1470  0.9693  0.1654
 -0.5769 -0.4730  0.3283  0.6729  1.3574 -0.6610  0.0265
  0.3767  1.0300  1.6927  0.4422  0.5837  1.5277  1.1686
  0.8843 -0.7698  0.0539 -0.3547  0.6904 -0.6842  0.2653
  0.4147  0.5062  0.6251  0.4374  0.3252  0.3478  0.0046
  0.7845 -0.0902  0.3499  0.0342  1.0706 -0.0605  0.5525

(2,.,.) =
 -0.7351 -0.9327 -0.3092 -1.3395 -0.4596 -0.6377 -0.5097
 -0.2406 -0.2617 -0.3400 -0.4339 -0.3648  0.1539 -0.2961
 -0.7124 -1.2228 -0.2632  0.1690  0.4836 -0.9469 -0.7003
 -0.0221  0.1067  0.6975 -0.4221 -0.3121  0.4822  0.6617
  0.2043 -0.9928 -0.9500 -1.6107  0.1409 -1.3548 -0.5212
 -0.3086 -0.0298 -0.2031  0.1026 -0.5785 -0.3275 -0.1630
  0.0596 -0.6097  0.1443 -0.8603 -0.2774 -0.4506 -0.5367

(3,.,.) =
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000 -0.7326  0.3544  0.1821  0.4796  1.0164  0.0000
  0.0000 -0.9195 -0.0567 -0.1947  0.0169  0.1924  0.0000
  0.0000  0.2596  0.6766  0.0939  0.5677  0.6359  0.0000
  0.0000 -0.2981 -1.2165 -0.0224 -1.1001  0.0008  0.0000
  0.0000 -0.1911  0.2912  0.5092  0.2955  0.7171  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

(4,.,.) =
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000 -0.8263  0.3646  0.6750  0.2062  0.2785  0.0000
  0.0000 -0.7572  0.0432 -0.0821  0.4871  1.9506  0.0000
  0.0000 -0.4609  0.4362  0.5091  0.8901 -0.6954  0.0000
  0.0000  0.6049 -0.1501 -0.4602 -0.6514  0.5439  0.0000
```

```
    0.0000  0.2570  0.4694 -0.1262  0.5602  0.0821  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

  (5,.,.) =
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.3158  0.4389 -0.0485 -0.2179  0.0000  0.0000
    0.0000  0.1966  0.6185 -0.9563 -0.3365  0.0000  0.0000
    0.0000 -0.2892 -0.9266 -0.0172 -0.3122  0.0000  0.0000
    0.0000 -0.6269  0.5349 -0.2520 -0.2187  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

  (6,.,.) =
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  1.1148  0.2324 -0.1093  0.5024  0.0000  0.0000
    0.0000 -0.2624 -0.5863  0.3444  0.3506  0.0000  0.0000
    0.0000  0.1486  0.8413  0.6229 -0.0130  0.0000  0.0000
    0.0000  0.8446  0.3801 -0.2611  0.8140  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  [torch.DoubleTensor of dimension 6x7x7]
```

Note how the last 2 of 6 filter maps have 1 column of zero-padding
on the left and top, as well as 2 on the right and bottom.
This is inevitable when the component
module output tensors non- `dim` sizes aren't all odd or even.
Such that in order to keep the mappings aligned, one need
only ensure that these be all odd (or even).

Bottle

```
module = nn.Bottle(module, [nInputDim], [nOutputDim])
```

Bottle allows varying dimensionality input to be forwarded through any module that accepts input of `nInputDim` dimensions, and generates output of `nOutputDim` dimensions.

Bottle can be used to forward a 4D input of varying sizes through a 2D module `b x n`. The module `Bottle(module, 2)` will accept input of shape `p x q x r x n` and outputs with the shape `p x q x r x m`. Internally Bottle will view the input of `module` as `p*q*r x n`, and view the output as `p x q x r x m`. The numbers `p x q x r` are inferred from the input and can change for every forward/backward pass.

```
input = torch.Tensor(4, 5, 3, 10)
mlp = nn.Bottle(nn.Linear(10, 2))

> print(input:size())
  4
  5
  3
 10
[torch.LongStorage of size 4]

> print(mlp:forward(input):size())
 4
 5
 3
 2
[torch.LongStorage of size 4]
```

Table Containers

While the above containers are used for manipulating input Tensors, table containers are used for manipulating tables :
* ConcatTable
* ParallelTable

These, along with all other modules for manipulating tables can be found here.