Table Layers

This set of modules allows the manipulation of `table`s through the layers of a neural network. This allows one to build very rich architectures:

* `table` Container Modules encapsulate sub-Modules:
  * `ConcatTable`: applies each member module to the same input `Tensor` and outputs a `table`;
  * `ParallelTable`: applies the `i`-th member module to the `i`-th input and outputs a `table`;
  * `MapTable`: applies a single module to every input and outputs a `table`;
* Table Conversion Modules convert between `table`s and `Tensor`s or `table`s:
  * `SplitTable`: splits a `Tensor` into a `table` of `Tensor`s;
  * `JoinTable`: joins a `table` of `Tensor`s into a `Tensor`;
  * `MixtureTable`: mixture of experts weighted by a gater;
  * `SelectTable`: select one element from a `table`;
  * `NarrowTable`: select a slice of elements from a `table`;
  * `FlattenTable`: flattens a nested `table` hierarchy;
* Pair Modules compute a measure like distance or similarity from a pair (`table`) of input `Tensor`s:
  * `PairwiseDistance`: outputs the `p`-norm. distance between inputs;
  * `DotProduct`: outputs the dot product (similarity) between inputs;
  * `CosineDistance`: outputs the cosine distance between inputs;
* CMath Modules perform element-wise operations on a `table` of `Tensor`s:
  * `CAddTable`: addition of input `Tensor`s;
  * `CSubTable`: substraction of input `Tensor`s;
  * `CMulTable`: multiplication of input `Tensor`s;
  * `CDivTable`: division of input `Tensor`s;
  * `CMaxTable`: max of input `Tensor`s;
  * `CMinTable`: min of input `Tensor`s;
* `Table` of Criteria:
  * `CriterionTable`: wraps a Criterion so that it can accept a `table` of inputs.

`table`-based modules work by supporting `forward()` and `backward()` methods that can accept `table`s as inputs.
It turns out that the usual `Sequential` module can do this, so all that is needed is other child modules that take advantage of such `table`s.

```
mlp = nn.Sequential()
t = {x, y, z}
pred = mlp:forward(t)
pred = mlp:forward{x, y, z}      -- This is equivalent to the line
before
```

ConcatTable

```
module = nn.ConcatTable()
```

`ConcatTable` is a container module that applies each member module to the same input `Tensor` or `table`.

```
                  +-----------+
             +----> {member1, |
+-------+    |    |           |
| input +----+---->  member2, |
+-------+    |    |           |
   or        +---->  member3} |
 {input}          +-----------+
```

Example 1

```
mlp = nn.ConcatTable()
mlp:add(nn.Linear(5, 2))
mlp:add(nn.Linear(5, 3))

pred = mlp:forward(torch.randn(5))
for i, k in ipairs(pred) do print(i, k) end
```

which gives the output:

```
1
```

```
-0.4073
 0.0110
[torch.Tensor of dimension 2]

2
 0.0027
-0.0598
-0.1189
[torch.Tensor of dimension 3]
```

Example 2

```
mlp = nn.ConcatTable()
mlp:add(nn.Identity())
mlp:add(nn.Identity())

pred = mlp:forward{torch.randn(2), {torch.randn(3)}}
print(pred)
```

which gives the output (using th):

```
{
  1 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
  2 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
}
```

ParallelTable

```
module = nn.ParallelTable()
```

`ParallelTable` is a container module that, in its `forward()` method, applies the `i`-th member module to the `i`-th input, and outputs a `table` of the set of outputs.

```
+----------+         +-----------+
| {input1, +---------> {member1, |
|          |         |           |
|  input2, +--------->  member2, |
|          |         |           |
|  input3} +--------->  member3} |
+----------+         +-----------+
```

Example

```
mlp = nn.ParallelTable()
mlp:add(nn.Linear(10, 2))
mlp:add(nn.Linear(5, 3))

x = torch.randn(10)
y = torch.rand(5)

pred = mlp:forward{x, y}
for i, k in pairs(pred) do print(i, k) end
```

which gives the output:

```
1
 0.0331
 0.7003
[torch.Tensor of dimension 2]

2
 0.0677
```

```
-0.1657
-0.7383
[torch.Tensor of dimension 3]
```

MapTable

```
module = nn.MapTable(m, share)
```

`MapTable` is a container for a single module which will be applied to all input elements. The member module is cloned as necessary to process all input elements. Call `resize(n)` to set the number of clones manually or call `clearState()` to discard all clones.

Optionally, the module can be initialized with the contained module and with a list of parameters that are shared across all clones. By default, these parameters are `weight`, `bias`, `gradWeight` and `gradBias`.

```
+----------+         +-----------+
| {input1, +---------> {member,  |
|          |         |           |
|  input2, +--------->  clone,   |
|          |         |           |
|  input3} +--------->  clone}   |
+----------+         +-----------+
```

Example

```
map = nn.MapTable()
map:add(nn.Linear(10, 3))

x1 = torch.rand(10)
x2 = torch.rand(10)
y = map:forward{x1, x2}

for i, k in pairs(y) do print(i, k) end
```

which gives the output:

```
1
 0.0345
 0.8695
 0.6502
[torch.DoubleTensor of size 3]

2
 0.0269
 0.4953
 0.2691
[torch.DoubleTensor of size 3]
```

SplitTable

```
module = SplitTable(dimension, nInputDims)
```

Creates a module that takes a `Tensor` as input and outputs several `table`s, splitting the `Tensor` along the specified `dimension`.
In the diagram below, `dimension` is equal to `1`.

```
    +----------+         +-----------+
    | input[1] +---------> {member1, |
  +----------+-+         |           |
  | input[2] +-----------> member2,  |
+----------+-+           |           |
| input[3] +-------------> member3}  |
+----------+             +-----------+
```

The optional parameter `nInputDims` allows to specify the number of dimensions that this module will receive.
This makes it possible to forward both minibatch and non-minibatch `Tensor`s through the same module.

Example 1

```
mlp = nn.SplitTable(2)
x = torch.randn(4, 3)
pred = mlp:forward(x)
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
 1.3885
 1.3295
 0.4281
-1.0171
[torch.Tensor of dimension 4]

2
-1.1565
-0.8556
-1.0717
-0.8316
[torch.Tensor of dimension 4]

3
-1.3678
-0.1709
-0.0191
-2.5871
[torch.Tensor of dimension 4]
```

Example 2

```
mlp = nn.SplitTable(1)
pred = mlp:forward(torch.randn(4, 3))
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
 1.6114
```

```
 0.9038
 0.8419
[torch.Tensor of dimension 3]

2
 2.4742
 0.2208
 1.6043
[torch.Tensor of dimension 3]

3
 1.3415
 0.2984
 0.2260
[torch.Tensor of dimension 3]

4
 2.0889
 1.2309
 0.0983
[torch.Tensor of dimension 3]
```

Example 3

```
mlp = nn.SplitTable(1, 2)
pred = mlp:forward(torch.randn(2, 4, 3))
for i, k in ipairs(pred) do print(i, k) end
pred = mlp:forward(torch.randn(4, 3))
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
-1.3533  0.7448 -0.8818
-0.4521 -1.2463  0.0316
[torch.DoubleTensor of dimension 2x3]

2
 0.1130 -1.3904  1.4620
 0.6722  2.0910 -0.2466
```

```
[torch.DoubleTensor of dimension 2x3]

3
 0.4672 -1.2738  1.1559
 0.4664  0.0768  0.6243
[torch.DoubleTensor of dimension 2x3]

4
 0.4194  1.2991  0.2241
 2.9786 -0.6715  0.0393
[torch.DoubleTensor of dimension 2x3]


1
-1.8932
 0.0516
-0.6316
[torch.DoubleTensor of dimension 3]

2
-0.3397
-1.8881
-0.0977
[torch.DoubleTensor of dimension 3]

3
 0.0135
 1.2089
 0.5785
[torch.DoubleTensor of dimension 3]

4
-0.1758
-0.0776
-1.1013
[torch.DoubleTensor of dimension 3]
```

The module also supports indexing from the end using negative dimensions. This allows to use this module when the number of dimensions of the input is unknown.

Example

```
m = nn.SplitTable(-2)
out = m:forward(torch.randn(3, 2))
for i, k in ipairs(out) do print(i, k) end
out = m:forward(torch.randn(1, 3, 2))
for i, k in ipairs(out) do print(i, k) end
```

gives the output:

```
1
 0.1420
-0.5698
[torch.DoubleTensor of size 2]

2
 0.1663
 0.1197
[torch.DoubleTensor of size 2]

3
 0.4198
-1.1394
[torch.DoubleTensor of size 2]


1
-2.4941
-1.4541
[torch.DoubleTensor of size 1x2]

2
 0.4594
 1.1946
[torch.DoubleTensor of size 1x2]

3
-2.3322
-0.7383
[torch.DoubleTensor of size 1x2]
```

A more complicated example

```lua
mlp = nn.Sequential()       -- Create a network that takes a Tensor
as input
mlp:add(nn.SplitTable(2))
c = nn.ParallelTable()      -- The two Tensor slices go through two
different Linear
c:add(nn.Linear(10, 3))     -- Layers in Parallel
c:add(nn.Linear(10, 7))
mlp:add(c)                  -- Outputing a table with 2 elements
p = nn.ParallelTable()      -- These tables go through two more
linear layers separately
p:add(nn.Linear(3, 2))
p:add(nn.Linear(7, 1))
mlp:add(p)
mlp:add(nn.JoinTable(1))    -- Finally, the tables are joined
together and output.

pred = mlp:forward(torch.randn(10, 2))
print(pred)

for i = 1, 100 do           -- A few steps of training such a
network..
   x = torch.ones(10, 2)
   y = torch.Tensor(3)
   y:copy(x:select(2, 1):narrow(1, 1, 3))
   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(0.05)

   print(err)
end
```

JoinTable

```lua
module = JoinTable(dimension, nInputDims)
```

Creates a module that takes a `table` of `Tensor` s as input and outputs a `Tensor` by joining them together along dimension `dimension` .
In the diagram below `dimension` is set to `1` .

```
+----------+             +-----------+
| {input1, +-------------> output[1] |
|          |           +-----------+-+
|  input2, +-----------> output[2] |
|          |         +-----------+-+
|  input3} +---------> output[3] |
+----------+         +-----------+
```

The optional parameter `nInputDims` allows to specify the number of dimensions that this module will receive. This makes it possible to forward both minibatch and non-minibatch `Tensor` s through the same module.

Example 1

```
x = torch.randn(5, 1)
y = torch.randn(5, 1)
z = torch.randn(2, 1)

print(nn.JoinTable(1):forward{x, y})
print(nn.JoinTable(2):forward{x, y})
print(nn.JoinTable(1):forward{x, z})
```

gives the output:

```
 1.3965
 0.5146
-1.5244
-0.9540
 0.4256
 0.1575
 0.4491
 0.6580
 0.1784
-1.7362
[torch.DoubleTensor of dimension 10x1]
```

```
 1.3965  0.1575
 0.5146  0.4491
-1.5244  0.6580
-0.9540  0.1784
 0.4256 -1.7362
[torch.DoubleTensor of dimension 5x2]

 1.3965
 0.5146
-1.5244
-0.9540
 0.4256
-1.2660
 1.0869
[torch.Tensor of dimension 7x1]
```

Example 2

```
module = nn.JoinTable(2, 2)

x = torch.randn(3, 1)
y = torch.randn(3, 1)

mx = torch.randn(2, 3, 1)
my = torch.randn(2, 3, 1)

print(module:forward{x, y})
print(module:forward{mx, my})
```

gives the output:

```
 0.4288  1.2002
-1.4084 -0.7960
-0.2091  0.1852
[torch.DoubleTensor of dimension 3x2]

(1,.,.) =
  0.5561  0.1228
 -0.6792  0.1153
  0.0687  0.2955
```

```
(2,.,.) =
  2.5787  1.8185
 -0.9860  0.6756
  0.1989 -0.4327
[torch.DoubleTensor of dimension 2x3x2]
```

A more complicated example

```
mlp = nn.Sequential()          -- Create a network that takes a
Tensor as input
c = nn.ConcatTable()           -- The same Tensor goes through two
different Linear
c:add(nn.Linear(10, 3))        -- Layers in Parallel
c:add(nn.Linear(10, 7))
mlp:add(c)                     -- Outputing a table with 2 elements
p = nn.ParallelTable()         -- These tables go through two more
linear layers
p:add(nn.Linear(3, 2))         -- separately.
p:add(nn.Linear(7, 1))
mlp:add(p)
mlp:add(nn.JoinTable(1))       -- Finally, the tables are joined
together and output.

pred = mlp:forward(torch.randn(10))
print(pred)

for i = 1, 100 do              -- A few steps of training such a
network..
   x = torch.ones(10)
   y = torch.Tensor(3); y:copy(x:narrow(1, 1, 3))
   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(0.05)

   print(err)
```

```
end
```

MixtureTable

`module` = `MixtureTable([dim])`

Creates a module that takes a `table` `{gater, experts}` as input and outputs the mixture of `experts` (a `Tensor` or `table` of `Tensor` s) using a `gater` `Tensor`. When `dim` is provided, it specifies the dimension of the `experts` `Tensor` that will be interpolated (or mixed). Otherwise, the `experts` should take the form of a `table` of `Tensor` s. This Module works for `experts` of dimension 1D or more, and for a 1D or 2D `gater`, i.e. for single examples or mini-batches.

Considering an `input = {G, E}` with a single example, then the mixture of experts `Tensor` `E` with gater `Tensor` `G` has the following form:

```
output = G[1]*E[1] + G[2]*E[2] + ... + G[n]*E[n]
```

where `dim = 1`, `n = E:size(dim) = G:size(dim)` and `G:dim() == 1`. Note that `E:dim() >= 2`, such that `output:dim() = E:dim() - 1`.

Example 1:
Using this Module, an arbitrary mixture of `n` 2-layer experts by a 2-layer gater could be constructed as follows:

```
experts = nn.ConcatTable()
for i = 1, n do
   local expert = nn.Sequential()
   expert:add(nn.Linear(3, 4))
   expert:add(nn.Tanh())
   expert:add(nn.Linear(4, 5))
   expert:add(nn.Tanh())
   experts:add(expert)
end

gater = nn.Sequential()
gater:add(nn.Linear(3, 7))
```

```
gater:add(nn.Tanh())
gater:add(nn.Linear(7, n))
gater:add(nn.SoftMax())

trunk = nn.ConcatTable()
trunk:add(gater)
trunk:add(experts)

moe = nn.Sequential()
moe:add(trunk)
moe:add(nn.MixtureTable())
```

Forwarding a batch of 2 examples gives us something like this:

```
> =moe:forward(torch.randn(2, 3))
-0.2152  0.3141  0.3280 -0.3772  0.2284
 0.2568  0.3511  0.0973 -0.0912 -0.0599
[torch.DoubleTensor of dimension 2x5]
```

Example 2:
In the following, the `MixtureTable` expects `experts` to be a `Tensor` of `size = {1, 4, 2, 5, n}`:

```
experts = nn.Concat(5)
for i = 1, n do
   local expert = nn.Sequential()
   expert:add(nn.Linear(3, 4))
   expert:add(nn.Tanh())
   expert:add(nn.Linear(4, 4*2*5))
   expert:add(nn.Tanh())
   expert:add(nn.Reshape(4, 2, 5, 1))
   experts:add(expert)
end

gater = nn.Sequential()
gater:add(nn.Linear(3, 7))
gater:add(nn.Tanh())
gater:add(nn.Linear(7, n))
gater:add(nn.SoftMax())

trunk = nn.ConcatTable()
trunk:add(gater)
trunk:add(experts)
```

```
moe = nn.Sequential()
moe:add(trunk)
moe:add(nn.MixtureTable(5))
```

Forwarding a batch of 2 examples gives us something like this:

```
> =moe:forward(torch.randn(2, 3)):size()
 2
 4
 2
 5
[torch.LongStorage of size 4]
```

SelectTable

`module` = `SelectTable(index)`

Creates a module that takes a (nested) `table` as input and outputs the element at index `index`. `index` can be strings or integers (positive or negative).
This can be either a `table` or a `Tensor`.

The gradients of the non- `index` elements are zeroed `Tensor` s of the same size. This is true regardless of the
depth of the encapsulated `Tensor` as the function used internally to do so is recursive.

Example 1:

```
> input = {torch.randn(2, 3), torch.randn(2, 1)}
> =nn.SelectTable(1):forward(input)
-0.3060  0.1398  0.2707
 0.0576  1.5455  0.0610
[torch.DoubleTensor of dimension 2x3]

> =nn.SelectTable(-1):forward(input)
 2.3080
-0.2955
[torch.DoubleTensor of dimension 2x1]
```

```
> =table.unpack(nn.SelectTable(1):backward(input, torch.randn(2,
3)))
-0.4891 -0.3495 -0.3182
-2.0999  0.7381 -0.5312
[torch.DoubleTensor of dimension 2x3]

0
0
[torch.DoubleTensor of dimension 2x1]
```

Exmaple 2:

```
> input = { A=torch.randn(2, 3), B=torch.randn(2, 1) }
> =nn.SelectTable("A"):forward(input)
-0.3060  0.1398  0.2707
 0.0576  1.5455  0.0610
[torch.DoubleTensor of dimension 2x3]

> gradInput = nn.SelectTable("A"):backward(input, torch.randn(2,
3))

> gradInput
{
  A : DoubleTensor - size: 2x3
  B : DoubleTensor - size: 2x1
}

> gradInput["A"]
-0.4891 -0.3495 -0.3182
-2.0999  0.7381 -0.5312
[torch.DoubleTensor of dimension 2x3]

> gradInput["B"]
0
0
[torch.DoubleTensor of dimension 2x1]
```

Example 3:

```
> input = {torch.randn(2, 3), {torch.randn(2, 1), {torch.randn(2,
2)}}}

> =nn.SelectTable(2):forward(input)
```

```
{
  1 : DoubleTensor - size: 2x1
  2 :
    {
      1 : DoubleTensor - size: 2x2
    }
}

> =table.unpack(nn.SelectTable(2):backward(input, {torch.randn(2,
1), {torch.randn(2, 2)}}))
0 0 0
0 0 0
[torch.DoubleTensor of dimension 2x3]

{
  1 : DoubleTensor - size: 2x1
  2 :
    {
      1 : DoubleTensor - size: 2x2
    }
}

> gradInput = nn.SelectTable(1):backward(input, torch.randn(2, 3))

> =gradInput
{
  1 : DoubleTensor - size: 2x3
  2 :
    {
      1 : DoubleTensor - size: 2x1
      2 :
        {
          1 : DoubleTensor - size: 2x2
        }
    }
}

> =gradInput[1]
-0.3400 -0.0404  1.1885
 1.2865  0.4107  0.6506
[torch.DoubleTensor of dimension 2x3]

> gradInput[2][1]
0
0
```

```
[torch.DoubleTensor of dimension 2x1]

> gradInput[2][2][1]
0 0
0 0
[torch.DoubleTensor of dimension 2x2]
```

NarrowTable

`module` = `NarrowTable(offset [, length])`

Creates a module that takes a `table` as input and outputs the subtable
starting at index `offset` having `length` elements (defaults to 1 element).
The elements can be either a `table` or a `Tensor`.

The gradients of the elements not included in the subtable are zeroed `Tensor`s of the same size.
This is true regardless of the depth of the encapsulated `Tensor` as the function used internally to do so is recursive.

Example:

```
> input = {torch.randn(2, 3), torch.randn(2, 1), torch.randn(1, 2)}
> =nn.NarrowTable(2,2):forward(input)
{
  1 : DoubleTensor - size: 2x1
  2 : DoubleTensor - size: 1x2
}

> =nn.NarrowTable(1):forward(input)
{
  1 : DoubleTensor - size: 2x3
}

> =table.unpack(nn.NarrowTable(1,2):backward(input, {torch.randn(2,
3), torch.randn(2, 1)}))
 1.9528 -0.1381  0.2023
 0.2297 -1.5169 -1.1871
[torch.DoubleTensor of size 2x3]
```

```
-1.2023
-0.4165
[torch.DoubleTensor of size 2x1]

 0  0
[torch.DoubleTensor of size 1x2]
```

FlattenTable

`module` = `FlattenTable()`

Creates a module that takes an arbitrarily deep `table` of `Tensor` s (potentially nested) as input and outputs a `table` of `Tensor` s, where the output `Tensor` in index `i` is the `Tensor` with post-order DFS index `i` in the input `table` .

This module is particularly useful in combination with nn.Identity() to create networks that can append to their input `table` .

Example:

```
x = {torch.rand(1), {torch.rand(2), {torch.rand(3)}},
torch.rand(4)}
print(x)
print(nn.FlattenTable():forward(x))
```

gives the output:

```
{
  1 : DoubleTensor - size: 1
  2 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
  3 : DoubleTensor - size: 4
}
```

```
{
  1 : DoubleTensor - size: 1
  2 : DoubleTensor - size: 2
  3 : DoubleTensor - size: 3
  4 : DoubleTensor - size: 4
}
```

PairwiseDistance

`module` = `PairwiseDistance(p)` creates a module that takes a `table` of two vectors as input and outputs the distance between them using the `p` -norm.

Example:

```
mlp_l1 = nn.PairwiseDistance(1)
mlp_l2 = nn.PairwiseDistance(2)
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp_l1:forward({x, y}))
print(mlp_l2:forward({x, y}))
```

gives the output:

```
 9
[torch.Tensor of dimension 1]

 5.1962
[torch.Tensor of dimension 1]
```

A more complicated example:

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp= nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other
-- so we make another copy of it called p2_mlp
-- this *shares* the same weights via the set command, but has its
```

```lua
own set of temporary gradient storage
-- that's why we create it again (so that the gradients of the pair
don't wipe each other)
p2_mlp= nn.Sequential(); p2_mlp:add(nn.Linear(5, 2))
p2_mlp:get(1).weight:set(p1_mlp:get(1).weight)
p2_mlp:get(1).bias:set(p1_mlp:get(1).bias)

-- we make a parallel table that takes a pair of examples as input.
they both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table and computes the pairwise distance between
-- the pair of outputs
mlp= nn.Sequential()
mlp:add(prl)
mlp:add(nn.PairwiseDistance(1))

-- and a criterion for pushing together or pulling apart pairs
crit = nn.HingeEmbeddingCriterion(1)

-- lets make two example vectors
x = torch.rand(5)
y = torch.rand(5)


-- Use a typical generic gradient update function
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

-- push the pair x and y together, notice how then the distance
between them given
-- by  print(mlp:forward({x, y})[1]) gets smaller
for i = 1, 10 do
gradUpdate(mlp, {x, y}, 1, crit, 0.01)
print(mlp:forward({x, y})[1])
end
```

```
-- pull apart the pair x and y, notice how then the distance
between them given
-- by  print(mlp:forward({x, y})[1]) gets larger

for i = 1, 10 do
gradUpdate(mlp, {x, y}, -1, crit, 0.01)
print(mlp:forward({x, y})[1])
end
```

DotProduct

`module` = `DotProduct()` creates a module that takes a `table` of two vectors (or matrices if in batch mode) as input and outputs the dot product between them.

Example:

```
mlp = nn.DotProduct()
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp:forward({x, y}))
```

gives the output:

```
 32
[torch.Tensor of dimension 1]
```

A more complicated example:

```
-- Train a ranking function so that mlp:forward({x, y}, {x, z})
returns a number
-- which indicates whether x is better matched with y or z (larger
score = better match), or vice versa.

mlp1 = nn.Linear(5, 10)
mlp2 = mlp1:clone('weight', 'bias')
```

```lua
prl = nn.ParallelTable();
prl:add(mlp1); prl:add(mlp2)

mlp1 = nn.Sequential()
mlp1:add(prl)
mlp1:add(nn.DotProduct())

mlp2 = mlp1:clone('weight', 'bias')

mlp = nn.Sequential()
prla = nn.ParallelTable()
prla:add(mlp1)
prla:add(mlp2)
mlp:add(prla)

x = torch.rand(5);
y = torch.rand(5)
z = torch.rand(5)


print(mlp1:forward{x, x})
print(mlp1:forward{x, y})
print(mlp1:forward{y, y})


crit = nn.MarginRankingCriterion(1);

-- Use a typical generic gradient update function
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

inp = {{x, y}, {x, z}}

math.randomseed(1)

-- make the pair x and y have a larger dot product than x and z

for i = 1, 100 do
```

```
    gradUpdate(mlp, inp, 1, crit, 0.05)
    o1 = mlp1:forward{x, y}[1];
    o2 = mlp2:forward{x, z}[1];
    o = crit:forward(mlp:forward{{x, y}, {x, z}}, 1)
    print(o1, o2, o)
end

print "________________**"

-- make the pair x and z have a larger dot product than x and y

for i = 1, 100 do
    gradUpdate(mlp, inp, -1, crit, 0.05)
    o1 = mlp1:forward{x, y}[1];
    o2 = mlp2:forward{x, z}[1];
    o = crit:forward(mlp:forward{{x, y}, {x, z}}, -1)
    print(o1, o2, o)
end
```

CosineDistance

`module` = `CosineDistance()` creates a module that takes a `table` of two vectors (or matrices if in batch mode) as input and outputs the cosine distance between them.

Examples:

```
mlp = nn.CosineDistance()
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp:forward({x, y}))
```

gives the output:

```
 0.9746
[torch.Tensor of dimension 1]
```

`CosineDistance` also accepts batches:

```
mlp = nn.CosineDistance()
x = torch.Tensor({{1,2,3},{1,2,-3}})
y = torch.Tensor({{4,5,6},{-4,5,6}})
print(mlp:forward({x,y}))
```

gives the output:

```
 0.9746
-0.3655
[torch.DoubleTensor of size 2]
```

A more complicated example:

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp= nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other
-- so we make another copy of it called p2_mlp
-- this *shares* the same weights via the set command, but has its
own set of temporary gradient storage
-- that's why we create it again (so that the gradients of the pair
don't wipe each other)
p2_mlp= p1_mlp:clone('weight', 'bias')

-- we make a parallel table that takes a pair of examples as input.
they both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table and computes the cosine distance between
-- the pair of outputs
mlp= nn.Sequential()
mlp:add(prl)
mlp:add(nn.CosineDistance())


-- lets make two example vectors
x = torch.rand(5)
```

```lua
y = torch.rand(5)

-- Grad update function..
function gradUpdate(mlp, x, y, learningRate)
    local pred = mlp:forward(x)
    if pred[1]*y < 1 then
        gradCriterion = torch.Tensor({-y})
        mlp:zeroGradParameters()
        mlp:backward(x, gradCriterion)
        mlp:updateParameters(learningRate)
    end
end

-- push the pair x and y together, the distance should get larger..
for i = 1, 1000 do
 gradUpdate(mlp, {x, y}, 1, 0.1)
 if ((i%100)==0) then print(mlp:forward({x, y})[1]);end
end


-- pull apart the pair x and y, the distance should get smaller..

for i = 1, 1000 do
 gradUpdate(mlp, {x, y}, -1, 0.1)
 if ((i%100)==0) then print(mlp:forward({x, y})[1]);end
end
```

CriterionTable

`module` = `CriterionTable(criterion)`

Creates a module that wraps a Criterion module so that it can accept a `table` of inputs. Typically the `table` would contain two elements: the input and output `x` and `y` that the Criterion compares.

Example:

```lua
mlp = nn.CriterionTable(nn.MSECriterion())
x = torch.randn(5)
y = torch.randn(5)
```

```
print(mlp:forward{x, x})
print(mlp:forward{x, y})
```

gives the output:

```
0
1.9028918413199
```

Here is a more complex example of embedding the criterion into a network:

```
function table.print(t)
 for i, k in pairs(t) do print(i, k); end
end

mlp = nn.Sequential();                          -- Create an mlp
that takes input
  main_mlp = nn.Sequential();             -- and output using
ParallelTable
  main_mlp:add(nn.Linear(5, 4))
  main_mlp:add(nn.Linear(4, 3))
 cmlp = nn.ParallelTable();
 cmlp:add(main_mlp)
 cmlp:add(nn.Identity())
mlp:add(cmlp)
mlp:add(nn.CriterionTable(nn.MSECriterion())) -- Apply the
Criterion

for i = 1, 20 do                                 -- Train for a few
iterations
 x = torch.ones(5);
 y = torch.Tensor(3); y:copy(x:narrow(1, 1, 3))
 err = mlp:forward{x, y}                         -- Pass in both
input and output
 print(err)

 mlp:zeroGradParameters();
 mlp:backward({x, y} );
 mlp:updateParameters(0.05);
end
```

CAddTable

`module` = `CAddTable([inplace])`

Takes a `table` of `Tensor` s and outputs summation of all `Tensor` s. If `inplace` is `true` , the sum is written to the first `Tensor` .

```
ii = {torch.ones(5), torch.ones(5)*2, torch.ones(5)*3}
=ii[1]
 1
 1
 1
 1
 1
[torch.DoubleTensor of dimension 5]

return ii[2]
 2
 2
 2
 2
 2
[torch.DoubleTensor of dimension 5]

return ii[3]
 3
 3
 3
 3
 3
[torch.DoubleTensor of dimension 5]

m = nn.CAddTable()
=m:forward(ii)
 6
 6
 6
 6
 6
[torch.DoubleTensor of dimension 5]
```

CSubTable

Takes a `table` with two `Tensor` and returns the component-wise subtraction between them.

```
m = nn.CSubTable()
=m:forward({torch.ones(5)*2.2, torch.ones(5)})
 1.2000
 1.2000
 1.2000
 1.2000
 1.2000
[torch.DoubleTensor of dimension 5]
```

CMulTable

Takes a `table` of `Tensor` s and outputs the multiplication of all of them.

```
ii = {torch.ones(5)*2, torch.ones(5)*3, torch.ones(5)*4}
m = nn.CMulTable()
=m:forward(ii)
 24
 24
 24
 24
 24
[torch.DoubleTensor of dimension 5]
```

CDivTable

Takes a `table` with two `Tensor` and returns the component-wise division between them.

```
m = nn.CDivTable()
=m:forward({torch.ones(5)*2.2, torch.ones(5)*4.4})
 0.5000
 0.5000
 0.5000
 0.5000
 0.5000
[torch.DoubleTensor of dimension 5]
```

CMaxTable

Takes a `table` of `Tensor` s and outputs the max of all of them.

```
m = nn.CMaxTable()
=m:forward({{torch.Tensor{1,2,3}, torch.Tensor{3,2,1}})
 3
 2
 3
[torch.DoubleTensor of size 3]
```

CMinTable

Takes a `table` of `Tensor` s and outputs the min of all of them.

```
m = nn.CMinTable()
=m:forward({{torch.Tensor{1,2,3}, torch.Tensor{3,2,1}})
 1
 2
 1
[torch.DoubleTensor of size 3]
```