Module

`Module` is an abstract class which defines fundamental methods necessary for a training a neural network. Modules are serializable.

Modules contain two states variables: output and gradInput.

[output] forward(input)

Takes an `input` object, and computes the corresponding `output` of the module. In general `input` and `output` are Tensors. However, some special sub-classes like table layers might expect something else. Please, refer to each module specification for further information.

After a `forward()`, the output state variable should have been updated to the new value.

It is not advised to override this function. Instead, one should implement updateOutput(input) function. The forward module in the abstract parent class Module will call `updateOutput(input)`.

[gradInput] backward(input, gradOutput)

Performs a *backpropagation step* through the module, with respect to the given `input`. In general this method makes the assumption forward(input) has been called before, *with the same input*. This is necessary for optimization reasons. If you do not respect this rule, `backward()` will compute incorrect gradients.

In general `input` and `gradOutput` and `gradInput` are Tensors. However, some special sub-classes like table layers might expect something else. Please, refer to each module specification for further information.

A *backpropagation step* consist in computing two kind of gradients

at `input` given `gradOutput` (gradients with respect to the output of the module). This function simply performs this task using two function calls:

- A function call to updateGradInput(input, gradOutput).
- A function call to accGradParameters(input,gradOutput,scale).

It is not advised to override this function call in custom classes. It is better to override updateGradInput(input, gradOutput) and accGradParameters(input, gradOutput,scale) functions.

updateOutput(input)

Computes the output using the current parameter set of the class and input. This function returns the result which is stored in the output field.

updateGradInput(input, gradOutput)

Computing the gradient of the module with respect to its own input. This is returned in `gradInput`. Also, the gradInput state variable is updated accordingly.

accGradParameters(input, gradOutput, scale)

Computing the gradient of the module with respect to its own parameters. Many modules do not perform this step as they do not have any parameters. The state variable name for the parameters is module dependent. The module is expected to *accumulate* the gradients with respect to the parameters in some variable.

`scale` is a scale factor that is multiplied with the gradParameters before being accumulated.

Zeroing this accumulation is achieved with zeroGradParameters() and updating

the parameters according to this accumulation is done with updateParameters().

zeroGradParameters()

If the module has parameters, this will zero the accumulation of the gradients with respect to these parameters, accumulated through accGradParameters(input, gradOutput,scale) calls. Otherwise, it does nothing.

updateParameters(learningRate)

If the module has parameters, this will update these parameters, according to the accumulation of the gradients with respect to these parameters, accumulated through backward() calls.

The update is basically:

```
parameters = parameters - learningRate * gradients_wrt_parameters
```

If the module does not have parameters, it does nothing.

accUpdateGradParameters(input, gradOutput, learningRate)

This is a convenience module that performs two functions at once. Calculates and accumulates the gradients with respect to the weights after multiplying with negative of the learning rate `learningRate`. Performing these two operations at once is more performance efficient and it might be advantageous in certain situations.

Keep in mind that, this function uses a simple trick to achieve its goal and it might not be valid for a custom module.

Also note that compared to accGradParameters(), the gradients are not retained for future use.

```
function Module:accUpdateGradParameters(input, gradOutput, lr)
   local gradWeight = self.gradWeight
   local gradBias = self.gradBias
   self.gradWeight = self.weight
   self.gradBias = self.bias
   self:accGradParameters(input, gradOutput, -lr)
   self.gradWeight = gradWeight
   self.gradBias = gradBias
end
```

As it can be seen, the gradients are accumulated directly into weights. This assumption may not be true for a module that computes a nonlinear operation.

share(mlp,s1,s2,...,sn)

This function modifies the parameters of the module named `s1`,.. `sn` (if they exist) so that they are shared with (pointers to) the parameters with the same names in the given module `mlp`.

The parameters have to be Tensors. This function is typically used if you want to have modules that share the same weights or biases.

Note that this function if called on a Container module will share the same parameters for all the contained modules as well.

Example:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a second mlp
mlp2=nn.Sequential();
mlp2:add(nn.Linear(100,10));

-- the second mlp shares the bias of the first
mlp2:share(mlp1,'bias');
```

```
-- we change the bias of the first
mlp1:get(1).bias[1]=99;

-- and see that the second one's bias has also changed..
print(mlp2:get(1).bias[1])
```

clone(mlp,...)

Creates a deep copy of (i.e. not just a pointer to) the module, including the current state of its parameters (e.g. weight, biases etc., if any).

If arguments are provided to the `clone(...)` function it also calls share(...) with those arguments on the cloned module after creating it, hence making a deep copy of this module with some shared parameters.

Example:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a copy that shares the weights and biases
mlp2=mlp1:clone('weight','bias');

-- we change the bias of the first mlp
mlp1:get(1).bias[1]=99;

-- and see that the second one's bias has also changed..
print(mlp2:get(1).bias[1])
```

type(type[, tensorCache])

This function converts all the parameters of a module to the given `type`. The `type` can be one of the types defined for torch.Tensor.

If tensors (or their storages) are shared between multiple modules in a

network, this sharing will be preserved after type is called.

To preserve sharing between multiple modules and/or tensors, use `nn.utils.recursiveType`:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a second mlp
mlp2=nn.Sequential();
mlp2:add(nn.Linear(100,10));

-- the second mlp shares the bias of the first
mlp2:share(mlp1,'bias');

-- mlp1 and mlp2 will be converted to float, and will share bias
-- note: tensors can be provided as inputs as well as modules
nn.utils.recursiveType({mlp1, mlp2}, 'torch.FloatTensor')
```

float([tensorCache])

Convenience method for calling module:type('torch.FloatTensor'[, tensorCache])

double([tensorCache])

Convenience method for calling module:type('torch.DoubleTensor'[, tensorCache])

cuda([tensorCache])

Convenience method for calling module:type('torch.CudaTensor'[, tensorCache])

State Variables

These state variables are useful objects if one wants to check the guts of

a `Module`. The object pointer is *never* supposed to change. However, its contents (including its size if it is a Tensor) are supposed to change.

In general state variables are
Tensors.
However, some special sub-classes
like table layers contain something else. Please,
refer to each module specification for further information.

output

This contains the output of the module, computed with the last call of forward(input).

gradInput

This contains the gradients with respect to the inputs of the module, computed with the last call of
updateGradInput(input, gradOutput).

Parameters and gradients w.r.t parameters

Some modules contain parameters (the ones that we actually want to train!). The name of these parameters, and gradients w.r.t these parameters are module dependent.

[{weights}, {gradWeights}] parameters()

This function should returns two tables. One for the learnable parameters `{weights}` and another for the gradients of the energy wrt to the learnable parameters `{gradWeights}`.

Custom modules should override this function if they use learnable parameters that are stored in tensors.

[flatParameters, flatGradParameters] getParameters()

This function returns two tensors. One for the flattened learnable parameters `flatParameters` and another for the gradients of the energy wrt to the learnable parameters `flatGradParameters`.

Custom modules should not override this function. They should instead override parameters(…) which is, in turn, called by the present function.

This function will go over all the weights and gradWeights and make them view into a single tensor (one for weights and one for gradWeights). Since the storage of every weight and gradWeight is changed, this function should be called only once on a given network.

training()

This sets the mode of the Module (or sub-modules) to `train=true`. This is useful for modules like Dropout or BatchNormalization that have a different behaviour during training vs evaluation.

evaluate()

This sets the mode of the Module (or sub-modules) to `train=false`. This is useful for modules like Dropout or BatchNormalization that have a different behaviour during training vs evaluation.

findModules(typename)

Find all instances of modules in the network of a certain `typename`. It returns a flattened list of the matching nodes, as well as a flattened list of the container modules for each matching node.

Modules that do not have a parent container (ie, a top level nn.Sequential for instance) will return their `self` as the container.

This function is very helpful for navigating complicated nested networks. For example, a didactic example might be; if you wanted to print the output size of all `nn.SpatialConvolution` instances:

```
-- Construct a multi-resolution convolution network (with 2
resolutions):
```

```
model = nn.ParallelTable()
conv_bank1 = nn.Sequential()
conv_bank1:add(nn.SpatialConvolution(3,16,5,5))
conv_bank1:add(nn.Threshold())
model:add(conv_bank1)
conv_bank2 = nn.Sequential()
conv_bank2:add(nn.SpatialConvolution(3,16,5,5))
conv_bank2:add(nn.Threshold())
model:add(conv_bank2)
-- FPROP a multi-resolution sample
input = {torch.rand(3,128,128), torch.rand(3,64,64)}
model:forward(input)
-- Print the size of the Threshold outputs
conv_nodes = model:findModules('nn.SpatialConvolution')
for i = 1, #conv_nodes do
  print(conv_nodes[i].output:size())
end
```

Another use might be to replace all nodes of a certain `typename` with another. For instance, if we wanted to replace all `nn.Threshold` with `nn.Tanh` in the model above:

```
threshold_nodes, container_nodes =
model:findModules('nn.Threshold')
for i = 1, #threshold_nodes do
  -- Search the container for the current threshold node
  for j = 1, #(container_nodes[i].modules) do
    if container_nodes[i].modules[j] == threshold_nodes[i] then
      -- Replace with a new instance
      container_nodes[i].modules[j] = nn.Tanh()
    end
  end
end
```

listModules()

List all Modules instances in a network. Returns a flattened list of modules, including container modules (which will be listed first), self, and any other component modules.

For example :

```
mlp = nn.Sequential()
mlp:add(nn.Linear(10,20))
mlp:add(nn.Tanh())
mlp2 = nn.Parallel()
mlp2:add(mlp)
mlp2:add(nn.ReLU())
for i,module in ipairs(mlp2:listModules()) do
   print(module)
end
```

Which will result in the following output :

```
nn.Parallel {
  input
    |`-> (1): nn.Sequential {
    |      [input -> (1) -> (2) -> output]
    |      (1): nn.Linear(10 -> 20)
    |      (2): nn.Tanh
    |    }
    |`-> (2): nn.ReLU
     ... -> output
}
nn.Sequential {
  [input -> (1) -> (2) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Tanh
}
nn.Linear(10 -> 20)
nn.Tanh
nn.ReLU
```

clearState()

Clears intermediate module states as `output`, `gradInput` and others.
Useful when serializing networks and running low on memory. Internally calls `set()` on tensors so it does not break buffer sharing.

apply(function)

Calls provided function on itself and all child modules. This function takes module to operate on as a first argument:

```
model:apply(function(module)
   module.train = true
end)
```

In the example above `train` will be set to to `true` in all modules of `model`. This is how `training()` and `evaluate()` functions implemented.

replace(function)

Similar to apply takes a function which applied to all modules of a model, but uses return value to replace the module. Can be used to replace all modules of one type to another or remove certain modules.

For example, can be used to remove `nn.Dropout` layers by replacing them with `nn.Identity`:

```
model:replace(function(module)
   if torch.typename(module) == 'nn.Dropout' then
      return nn.Identity()
   else
      return module
   end
end)
```