dpnn : deep extensions to nn

This package provides many useful features that aren't part of the main nn package.
These include sharedClone, which allows you to clone a module and share parameters or gradParameters with the original module, without incuring any memory overhead.
We also redefined type such that the type-cast preserves Tensor sharing within a structure of modules.

The package provides the following Modules:

- Decorator : abstract class to change the behaviour of an encapsulated module ;
- DontCast : prevent encapsulated module from being casted by `Module:type()` ;
- Serial : decorate a module makes its serialized output more compact ;
- NaN : decorate a module to detect the source of NaN errors ;
- Inception : implements the Inception module of the GoogleLeNet article ;
- Collapse : just like `nn.View(-1)` ;
- Convert : convert between different tensor types or shapes;
- ZipTable : zip a table of tables into a table of tables;
- ZipTableOneToMany : zip a table of element `el` and table of elements into a table of pairs of element `el` and table elements;
- CAddTensorTable : adds a tensor to a table of tensors of the same size;
- ReverseTable : reverse the order of elements in a table;
- PrintSize : prints the size of inputs and gradOutputs (useful for debugging);
- Clip : clips the inputs to a min and max value;
- Constant : outputs a constant value given an input (which is ignored);
- SpatialUniformCrop : uniformly crops patches from a input;
- SpatialGlimpse : takes a fovead glimpse of an image at a given location;
- WhiteNoise : adds isotropic Gaussian noise to the signal when in training mode;
- OneHot : transforms a tensor of indices into one-hot encoding;
- Kmeans : Kmeans clustering layer. Forward computes distances with respect to centroids and returns index of closest centroid. Centroids can be updated using gradient descent. Centroids could be initialized randomly or by using kmeans++ algoirthm;
- SpatialRegionDropout : Randomly dropouts a region (top, bottom, leftmost, rightmost) of the input image. Works with batch and any number of channels;
- FireModule : FireModule as mentioned in the SqueezeNet;
- NCEModule : optimized placeholder for a `Linear` + `SoftMax` using noise-contrastive estimation.
- SpatialFeatNormalization : Module for widely used preprocessing step of mean zeroing and standardization for images.
- SpatialBinaryConvolution : Module for binary spatial convolution (Binary weights) as

mentioned in XNOR-Net.
- SimpleColorTransform : Module for adding independent random noise to input image channels.
- PCAColorTransform : Module for adding noise to input image using Principal Components Analysis.

The following modules and criterions can be used to implement the REINFORCE algorithm :

- Reinforce : abstract class for REINFORCE modules;
- ReinforceBernoulli : samples from Bernoulli distribution;
- ReinforceNormal : samples from Normal distribution;
- ReinforceGamma : samples from Gamma distribution;
- ReinforceCategorical : samples from Categorical (Multinomial with one sample) distribution;
- VRClassReward : criterion for variance-reduced classification-based reward;
- BinaryClassReward : criterion for variance-reduced binary classification reward (like `VRClassReward`, but for binary classes);

Additional differentiable criterions
* BinaryLogisticRegression : criterion for binary logistic regression;
* SpatialBinaryLogisticRegression : criterion for pixel wise binary logistic regression;
* NCECriterion : criterion exclusively used with NCEModule.
* ModuleCriterion : adds an optional `inputModule` and `targetModule` before a decorated criterion;
* BinaryLogisticRegression : criterion for binary logistic regression.
* SpatialBinaryLogisticRegression : criterion for pixel wise binary logistic regression.

A lot of the functionality implemented here was pulled from
dp, which makes heavy use of this package.
However, dpnn can be used without dp (for e.g. you can use it with optim),
which is one of the main reasons why we made it.

Tutorials

Sagar Waghmare wrote a nice tutorial
on how to use dpnn with nngraph to reproduce the
Lateral Connections in Denoising Autoencoders Support Supervised Learning.

A brief (1 hours) overview of Torch7, which includes some details about **dpnn**,
is available via this NVIDIA GTC Webinar video. In any case, this presentation gives a nice
overview of Logistic Regression, Multi-Layer Perceptrons, Convolutional Neural Networks and

Recurrent Neural Networks using Torch7.

Module

The Module interface has been further extended with methods that facilitate stochastic gradient descent like updateGradParameters (i.e. momentum learning), weightDecay, maxParamNorm (for regularization), and so on.

Module.dpnn_parameters

A table that specifies the name of parameter attributes.
Defaults to `{'weight', 'bias'}`, which is a static variable (i.e. table exists in class namespace).
Sub-classes can define their own table statically.

Module.dpnn_gradParameters

A table that specifies the name of gradient w.r.t. parameter attributes.
Defaults to `{'gradWeight', 'gradBias'}`, which is a static variable (i.e. table exists in class namespace).
Sub-classes can define their own table statically.

[self] Module:type(type_str)

This function converts all the parameters of a module to the given `type_str`.
The `type_str` can be one of the types defined for torch.Tensor
like `torch.DoubleTensor`, `torch.FloatTensor` and `torch.CudaTensor`.
Unlike the type method
defined in nn, this one was overriden to
maintain the sharing of storage
among Tensors. This is especially useful when cloning modules share `parameters` and `gradParameters`.

[clone] Module:sharedClone([shareParams, shareGradParams])

Similar to clone.
Yet when `shareParams = true` (the default), the cloned module will share the parameters with the original module.
Furthermore, when `shareGradParams = true` (the default), the clone module will share the gradients w.r.t. parameters with the original module.
This is equivalent to :

```
clone = mlp:clone()
clone:share(mlp, 'weight', 'bias', 'gradWeight', 'gradBias')
```

yet it is much more efficient, especially for modules with lots of parameters, as these Tensors aren't needlessly copied during the `clone`.
This is particularly useful for Recurrent neural networks
which require efficient copies with shared parameters and gradient w.r.t. parameters for each time-step.

Module:maxParamNorm([maxOutNorm, maxInNorm])

This method implements a hard constraint on the upper bound of the norm of output and/or input neuron weights
(Hinton et al. 2012, p. 2) .
In a weight matrix, this is a contraint on rows (`maxOutNorm`) and/or columns (`maxInNorm`), respectively.
Has a regularization effect analogous to weightDecay, but with easier to optimize hyper-parameters.
Assumes that parameters are arranged (`output dim x ... x input dim`).
Only affects parameters with more than one dimension.
The method should normally be called after updateParameters.
It uses the C/CUDA optimized torch.renorm function.
Hint : `maxOutNorm = 2` usually does the trick.

[momGradParams] Module:momentumGradParameters()

Returns a table of Tensors (`momGradParams`). For each element in the table, a corresponding parameter (`params`) and gradient w.r.t. parameters (`gradParams`) is returned by a call to parameters.
This method is used internally by updateGradParameters.

Module:updateGradParameters(momFactor [, momDamp, momNesterov])

Applies classic momentum or Nesterov momentum (Sutskever, Martens et al, 2013) to parameter gradients.
Each parameter Tensor (`params`) has a corresponding Tensor of the same size for gradients w.r.t. parameters (`gradParams`).
When using momentum learning, another Tensor is added for each parameter Tensor (`momGradParams`).
This method should be called before updateParameters
as it affects the gradients w.r.t. parameters.

Classic momentum is computed as follows :

```
momGradParams = momFactor*momGradParams + (1-momDamp)*gradParams
gradParams = momGradParams
```

where `momDamp` has a default value of `momFactor` .

Nesterov momentum (`momNesterov = true`) is computed as follows (the first line is the same as classic momentum):

```
momGradParams = momFactor*momGradParams + (1-momDamp)*gradParams
gradParams = gradParams + momFactor*momGradParams
```

The default is to use classic momentum (`momNesterov = false`).

Module:weightDecay(wdFactor [, wdMinDim])

Decays the weight of the parameterized models.
Implements an L2 norm loss on parameters with dimensions greater or equal to `wdMinDim` (default is 2).

The resulting gradients are stored into the corresponding gradients w.r.t. parameters.
Such that this method should be called before updateParameters.

Module:gradParamClip(cutoffNorm [, moduleLocal])

Implements a contrainst on the norm of gradients w.r.t. parameters (Pascanu et al. 2012).
When `moduleLocal = false` (the default), the norm is calculated globally to Module for which this is called.
So if you call it on an MLP, the norm is computed on the concatenation of all parameter Tensors.
When `moduleLocal = true`, the norm constraint is applied
to the norm of all parameters in each component (non-container) module.
This method is useful to prevent the exploding gradient in
Recurrent neural networks.

Module:reinforce(reward)

This method is used by Criterions that implement the REINFORCE algorithm like VRClassReward.
While vanilla backpropagation (gradient descent using the chain rule),
REINFORCE Criterions broadcast a `reward` to all REINFORCE modules between the `forward` and the `backward`.
In this way, when the following call to `backward` reaches the REINFORCE modules,
these will compute a `gradInput` using the broadcasted `reward`.
The `reward` is broadcast to all REINFORCE modules contained
within `model` by calling `model:reinforce(reward)`.
Note that the `reward` should be a 1D tensor of size `batchSize`,
i.e. each example in a batch has its own scalar reward.

Refer to this example
for a complete training script making use of the REINFORCE interface.

Decorator

```
dmodule = nn.Decorator(module)
```

This module is an abstract class used to decorate a `module`. This means that method calls to `dmodule` will call the same method on the encapsulated `module`, and return its results.

DontCast

```
dmodule = nn.DontCast(module)
```

This module is a decorator. Use it to decorate a module that you don't want to be cast when the `type()` method is called.

```
module = nn.DontCast(nn.Linear(3,4):float())
module:double()
th> print(module:forward(torch.FloatTensor{1,2,3}))
 1.0927
-1.9380
-1.8158
-0.0805
[torch.FloatTensor of size 4]
```

Serial

```
dmodule = nn.Serial(module, [tensortype])
dmodule:[light,medium,heavy]Serial()
```

This module is a decorator that can be used to control the serialization/deserialization behavior of the encapsulated module. Basically, making the resulting string or file heavy (the default), medium or light in terms of size.

Furthermore, when specified, the `tensortype` attribute (e.g *torch.FloatTensor*, *torch.DoubleTensor* and so on.),
determines what type the module will be cast to during serialization.
Note that this will also be the type of the deserialized object.

The default serialization `tensortype` is `nil`, i.e. the module is serialized as is.

The `heavySerial()` has the serialization process serialize every attribute in the module graph,
which is the default behavior of nn.

The `mediumSerial()` has the serialization process serialize
everything except the attributes specified in each module's `dpnn_mediumEmpty`
table, which has a default value of `{'output', 'gradInput', 'momGradParams', 'dpnn_input'}`.
During serialization, whether they be tables or Tensors, these attributes are emptied (no storage).
Some modules overwrite the default `Module.dpnn_mediumEmpty` static attribute with their own.

The `lightSerial()` has the serialization process empty
everything a call to `mediumSerial(type)` would (so it uses `dpnn_mediumEmpty`).
But also empties all the parameter gradients specified by the
attribute `dpnn_gradParameters`, which defaults to `{gradWeight, gradBias}`.

We recomment using `mediumSerial()` for training, and `lightSerial()` for production (feed-forward-only models).

NaN

```
dmodule = nn.NaN(module, [id])
```

The `NaN` module asserts that the `output` and `gradInput` of the decorated `module` do not contain NaNs.
This is useful for locating the source of those pesky NaN errors.
The `id` defaults to automatically incremented values of `1,2,3,...`.

For example:

```
linear = nn.Linear(3,4)
mlp = nn.Sequential()
mlp:add(nn.NaN(nn.Identity()))
mlp:add(nn.NaN(linear))
mlp:add(nn.NaN(nn.Linear(4,2)))
print(mlp)
```

As you can see the `NaN` layers are have unique ids :

```
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.NaN(1) @ nn.Identity
  (2): nn.NaN(2) @ nn.Linear(3 -> 4)
  (3): nn.NaN(3) @ nn.Linear(4 -> 2)
}
```

And if we fill the `bias` of the linear module with NaNs and call `forward` :

```
nan = math.log(math.log(0)) -- this is a nan value
linear.bias:fill(nan)
mlp:forward(torch.randn(2,3))
```

We get a nice error message:

```
/usr/local/share/lua/5.1/dpnn/NaN.lua:39: NaN found in parameters
of module :
nn.NaN(2) @ nn.Linear(3 -> 4)
```

Inception

References :

- A. Going Deeper with Convolutions
- B. GoogleLeNet

```
module = nn.Inception(config)
```

This module uses `n` +2 parallel “columns”.
The original paper uses 2+2 where the first two are (but there could be more than two):

- 1x1 conv (reduce) -> relu -> 5x5 conv -> relu
- 1x1 conv (reduce) -> relu -> 3x3 conv -> relu

and where the other two are :

- 3x3 maxpool -> 1x1 conv (reduce/project) -> relu
- 1x1 conv (reduce) -> relu.

This module allows the first group of columns to be of any
number while the last group consist of exactly two columns.
The 1x1 convoluations are used to reduce the number of input channels
(or filters) such that the capacity of the network doesn't explode.
We refer to these here has *reduce*.
Since each column seems to have one and only one reduce, their initial
configuration options are specified in lists of n+2 elements.

The sole argument `config` is a table taking the following key-values :

- Required Arguments :
  - `inputSize` : number of input channels or colors, e.g. 3;
  - `outputSize` : numbers of filters in the non-1x1 convolution kernel sizes, e.g. `{32,48}`
  - `reduceSize` : numbers of filters in the 1x1 convolutions (reduction) used in each column, e.g. `{48,64,32,32}` . The last 2 are used respectively for the max pooling (projection) column (the last column in the paper) and the column that has nothing but a 1x1 conv (the first column in the paper). This table should have two elements more than the outputSize

- Optional Arguments :
  - `reduceStride` : strides of the 1x1 (reduction) convolutions. Defaults to `{1,1,...}` .
  - `transfer` : transfer function like `nn.Tanh` , `nn.Sigmoid` , `nn.ReLU` , `nn.Identity` , etc. It is used after each reduction (1x1 convolution) and convolution. Defaults to `nn.ReLU` .
  - `batchNorm` : set this to `true` to use batch normalization. Defaults to `false` . Note that batch normalization can be awesome
  - `padding` : set this to `true` to add padding to the input of the convolutions such that output width and height are same as that of the original non-padded `input` . Defaults to `true` .
  - `kernelSize` : size (`height = width`) of the non-1x1 convolution kernels. Defaults to `{5,3}` .
  - `kernelStride` : stride of the kernels (`height = width`) of the convolution. Defaults to `{1,1}`
  - `poolSize` : size (`height = width`) of the spatial max pooling used in the next-to-last column. Defaults to 3.
  - `poolStride` : stride (`height = width`) of the spatial max pooling. Defaults to 1.

For a complete example using this module, refer to the following :
* deep inception training script ;
* openface facial recognition (the model definition is here).

Collapse

```
module = nn.Collapse(nInputDim)
```

This module is the equivalent of:

```
view = nn.View(-1)
view:setNumInputDim(nInputDim)
```

It collapses all non-batch dimensions. This is useful for converting
a spatial feature map to the single dimension required by a dense
hidden layer like Linear.

Convert

```
module = nn.Convert([inputShape, outputShape])
```

Module to convert between different data formats.
For example, we can flatten images by using :

```
module = nn.Convert('bchw', 'bf')
```

or equivalently

```
module = nn.Convert('chw', 'f')
```

Lets try it with an input:

```
print(module:forward(torch.randn(3,2,3,1)))
 0.5692 -0.0190  0.5243  0.7530  0.4230  1.2483
-0.9142  0.6013  0.5608 -1.0417 -1.4014  1.0177
-1.5207 -0.1641 -0.4166  1.4810 -1.1725 -1.0037
[torch.DoubleTensor of size 3x6]
```

You could also try:

```
module = nn.Convert('chw', 'hwc')
input = torch.randn(1,2,3,2)
input:select(2,1):fill(1)
input:select(2,2):fill(2)
print(input)
(1,1,.,.) =
  1  1
  1  1
  1  1
(1,2,.,.) =
  2  2
  2  2
  2  2
[torch.DoubleTensor of size 1x2x3x2]
print(module:forward(input))
(1,1,.,.) =
  1  2
  1  2

(1,2,.,.) =
  1  2
  1  2

(1,3,.,.) =
  1  2
  1  2
[torch.DoubleTensor of size 1x3x2x2]
```

Furthermore, it automatically converts the `input` to have the same type as `self.output` (i.e. the type of the module).
So you can also just use is for automatic input type converions:

```
module = nn.Convert()
print(module.output) -- type of module
```

```
[torch.DoubleTensor with no dimension]
input = torch.FloatTensor{1,2,3}
print(module:forward(input))
 1
 2
 3
[torch.DoubleTensor of size 3]
```

ZipTable

```
module = nn.ZipTable()
```

Zips a table of tables into a table of tables.

Example:

```
print(module:forward{ {'a1','a2'}, {'b1','b2'}, {'c1','c2'} })
{ {'a1','b1','c1'}, {'a2','b2','c2'} }
```

ZipTableOneToMany

```
module = nn.ZipTableOneToMany()
```

Zips a table of element `el` and table of elements `tab` into a table of tables, where the i-th table contains the element `el` and the i-th element in table `tab`

Example:

```
print(module:forward{ 'el', {'a','b','c'} })
{ {'el','a'}, {'el','b'}, {'el','c'} }
```

CAddTensorTable

```
module = nn.CAddTensorTable()
```

Adds the first element `el` of the input table `tab` to each tensor contained in the second element of `tab`, which is itself a table

Example:

```
print(module:forward{ (0,1,1), {(0,0,0),(1,1,1)} })
{ (0,1,1), (1,2,2) }
```

ReverseTable

```
module = nn.ReverseTable()
```

Reverses the order of elements in a table.

Example:

```
print(module:forward{1,2,3,4})
{4,3,2,1}
```

PrintSize

```
module = nn.PrintSize(name)
```

This module is useful for debugging complicated module composites.
It prints the size of the `input` and `gradOutput` during `forward`

and `backward` propagation respectively.
The `name` is a string used to identify the module along side the printed size.

Clip

```
module = nn.Clip(minval, maxval)
```

This module clips `input` values such that the output is between `minval` and `maxval`.

Constant

```
module = nn.Constant(value, nInputDim)
```

This module outputs a constant value given an input.
If `nInputDim` is specified, it uses the input to determine the size of the batch.
The `value` is then replicated over the batch.
Otherwise, the `value` Tensor is output as is.
During `backward`, the returned `gradInput` is a zero Tensor of the same size as the `input`.
This module has no trainable parameters.

You can use this with nn.ConcatTable() to append constant inputs to an input :

```
nn.ConcatTable():add(nn.Constant(v)):add(nn.Identity())
```

This is useful when you want to output a value that is independent of the
input to the neural network (see this example).

SpatialUniformCrop

```
module = nn.SpatialUniformCrop(oheight, owidth)
```

During training, this module will output a cropped patch of size `oheight, owidth`
within the boundaries of the `input` image.
For each example, a location is sampled from a uniform distribution
such that each possible patch has an equal probability of being sampled.

During evaluation, the center patch is cropped and output.

This module is commonly used at the input layer to artificially
augment the size of the dataset to prevent overfitting.

SpatialGlimpse

Ref. A. Recurrent Model for Visual Attention

```
module = nn.SpatialGlimpse(size, depth, scale)
```

A glimpse is the concatenation of down-scaled cropped images of
increasing scale around a given location in a given image.
The input is a pair of Tensors: `{image, location}`
`location` are `(y,x)` coordinates of the center of the different scales
of patches to be cropped from image `image`.
Coordinates are between `(-1,-1)` (top-left) and `(1,1)` (bottom-right).
The `output` is a batch of glimpses taken in image at location `(y,x)`.

`size` can be either a scalar which specifies the `width = height` of glimpses,
or a table of `{height, width}` to support a rectangular shape of glimpses.
`depth` is number of patches to crop per glimpse (one patch per depth).
`scale` determines the `size(t) = scale * size(t-1)` of successive cropped patches.

So basically, this module can be used to focus the attention of the model
on a region of the input `image`.
It is commonly used with the RecurrentAttention
module (see this example).

WhiteNoise

```
module = nn.WhiteNoise([mean, stdev])
```

Useful in training [Denoising Autoencoders] (http://arxiv.org/pdf/1507.02672v1.pdf).
Takes `mean` and `stdev` of the normal distribution as input.
Default values for mean and standard deviation are 0 and 0.1 respectively.
With `module:training()`, noise is added during forward.
During `backward` gradients are passed as it is.
With `module:evaluate()` the mean is added to the input.

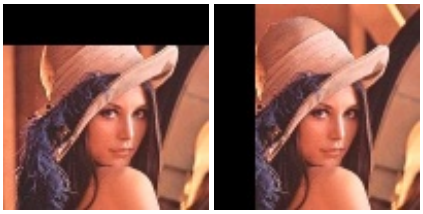
SpatialRegionDropout

```
module = nn.SpatialRegionDropout(p)
```

Following is an example of `SpatialRegionDropout` outputs on the famous lena image.

Input

Outputs



FireModule

Ref: http://arxiv.org/pdf/1602.07360v1.pdf

```
module = nn.FireModule(nInputPlane, s1x1, e1x1, e3x3, activation)
```

FireModule is comprised of two submodules 1) A *squeeze* convolution module comprised of `1x1` filters followed by 2) an *expand* module that is comprised of a mix of `1x1` and `3x3` convolution filters.
Arguments: `s1x1` : number of `1x1` filters in the squeeze submodule, `e1x1` : number of `1x1` filters in the expand submodule, `e3x3` : number of `3x3` filters in the expand submodule. It is recommended that `s1x1` be less than `(e1x1+e3x3)` if you want to limit the number of input channels to the `3x3` filters in the expand submodule.
FireModule works only with batches, for single sample convert the sample to a batch of size 1.

SpatialFeatNormalization

```
module = nn.SpatialFeatNormalization(mean, std)
```

This module normalizies each feature channel of input image based on its corresponding mean and standard deviation scalar values. This module does not learn the `mean` and `std`, they are provided as arguments.

SpatialBinaryConvolution

```
module = nn.SpatialBinaryConvolution(nInputPlane, nOutputPlane, kW,
kH)
```

Functioning of SpatialBinaryConvolution is similar to nn/SpatialConvolution. Only difference is that Binary weights are used for forward/backward and floating point weights are used for weight updates. Check **Binary-Weight-Network** section of XNOR-net.

SimpleColorTransform

```
range = torch.rand(inputChannels) -- Typically range is specified
by user.
module = nn.SimpleColorTransform(inputChannels, range)
```

This module performs a simple data augmentation technique. SimpleColorTransform module adds random noise to each color channel independently. In more advanced data augmentation technique noise is added using principal components of color channels. For that please check **PCAColorTransform**

PCAColorTransform

```
eigenVectors = torch.rand(inputChannels, inputChannels) -- Eigen
Vectors
eigenValues = torch.rand(inputChannels) -- Eigen
std = 0.1 -- Std deviation of normal distribution with mean zero
for noise.
module = nn.PCAColorTransform(inputChannels, eigenVectors,
eigenValues, std)
```

This module performs a data augmentation using Principal Component analysis of pixel values. When in training mode, mulitples of principal components are added to input image pixels. Magnitude of value added (noise) is dependent upon the corresponding eigen value and a random value sampled from a Gaussian distribution with mean zero and `std` (default 0.1) standard deviation. This technique was used in the famous AlexNet paper.

OneHot

```
module = nn.OneHot(outputSize)
```

Transforms a tensor of `input` indices having integer values between 1 and `outputSize` into a tensor of one-hot vectors of size `outputSize`.

Forward an index to get a one-hot vector :

```
> module = nn.OneHot(5) -- 5 classes
> module:forward(torch.LongTensor{3})
 0  0  1  0  0
[torch.DoubleTensor of size 1x5]
```

Forward a batch of 3 indices. Notice that these need not be stored as `torch.LongTensor` :

```
> module:forward(torch.Tensor{3,2,1})
 0  0  1  0  0
 0  1  0  0  0
 1  0  0  0  0
[torch.DoubleTensor of size 3x5]
```

Forward batch of `2 x 3` indices :

```
oh:forward(torch.Tensor{{3,2,1},{1,2,3}})
(1,.,.) =
  0  0  1  0  0
  0  1  0  0  0
  1  0  0  0  0

(2,.,.) =
  1  0  0  0  0
  0  1  0  0  0
  0  0  1  0  0
[torch.DoubleTensor of size 2x3x5]
```

Kmeans

```
km = nn.Kmeans(k, dim)
```

`k` is the number of centroids and `dim` is the dimensionality of samples.
You can either initialize centroids randomly from input samples or by using *kmeans++* algorithm.

```
km:initRandom(samples) -- Randomly initialize centroids from input
```

```
samples.
km:initKmeansPlus(samples) -- Use Kmeans++ to initialize centroids.
```

Example showing how to use Kmeans module to do standard Kmeans clustering.

```
attempts = 10
iter = 100 -- Number of iterations
bestKm = nil
bestLoss = math.huge
learningRate = 1
for j=1, attempts do
   local km = nn.Kmeans(k, dim)
   km:initKmeansPlus(samples)
   for i=1, iter do
      km:zeroGradParameters()
      km:forward(samples) -- sets km.loss
      km:backward(samples, gradOutput) -- gradOutput is ignored

      -- Gradient Descent weight/centroids update
      km:updateParameters(learningRate)
   end

   if km.loss < bestLoss then
      bestLoss = km.loss
      bestKm = km:clone()
   end
end
```

`nn.Kmeans()` module maintains loss only for the latest forward. If you want to maintain loss over the whole dataset then you who would need do it my adding the module loss for every forward.

You can also use `nn.Kmeans()` as an auxillary layer in your network.
A call to `forward` will generate an `output` containing the index of the nearest cluster for each sample in the batch.
The `gradInput` generated by `updateGradInput` will be zero.

ModuleCriterion

```
criterion = nn.ModuleCriterion(criterion [, inputModule,
targetModule, castTarget])
```

This criterion decorates a `criterion` by allowing the `input` and `target` to be fed through an optional `inputModule` and `targetModule` before being passed to the `criterion`. The `inputModule` must not contain parameters as these would not be updated.

When `castTarget = true` (the default), the `targetModule` is cast along with the `inputModule` and
`criterion`. Otherwise, the `targetModule` isn't.

NCEModule

Ref. A RNNLM training with NCE for Speech Recognition

```
ncem = nn.NCEModule(inputSize, outputSize, k, unigrams, [Z])
```

When used in conjunction with NCECriterion,
the `NCEModule` implements noise-contrastive estimation.

The point of the NCE is to speedup computation for large `Linear` + `SoftMax` layers.
Computing a forward/backward for `Linear(inputSize, outputSize)` for a large `outputSize` can be very expensive.
This is common when implementing language models having with large vocabularies of a million words.
In such cases, NCE can be an efficient alternative to computing the full `Linear` + `SoftMax` during training and
cross-validation.

The `inputSize` and `outputSize` are the same as for the `Linear` module.
The number of noise samples to be drawn per example is `k`. A value of 25 should work well.
Increasing it will yield better results, while a smaller value will be more efficient to process.
The `unigrams` is a tensor of size `outputSize` that contains the frequencies or probability distribution over classes.
It is used to sample noise samples via a fast implementation of `torch.multinomial`.
The `Z` is the normalization constant of the approximated SoftMax.
The default is `math.exp(9)` as specified in Ref. A.

For inference, or measuring perplexity, the full `Linear` + `SoftMax` will need to

be computed. The `NCEModule` can do this by switching on the following :

```
ncem:evaluate()
ncem.normalized = true
```

Furthermore, to simulate `Linear` + `LogSoftMax` instead, one need only add the following to the above:

```
ncem.logsoftmax = true
```

An example is provided via the rnn package.

NCECriterion

```
ncec = nn.NCECriterion()
```

This criterion only works with an NCEModule on the output layer.
Together, they implement noise-contrastive estimation.

Reinforce

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

Abstract class for modules that implement the REINFORCE algorithm (ref. A).

```
module = nn.Reinforce([stochastic])
```

The `reinforce(reward)` method is called by a special Reward Criterion (e.g. VRClassReward).
After which, when backward is called, the reward will be used to generate gradInputs.
When `stochastic=true`, the module is stochastic (i.e. samples from a distribution) during evaluation and training.

When `stochastic=false` (the default), the module is only stochastic during training.

The REINFORCE rule for a module can be summarized as follows :

```
              d ln(f(output,input))
gradInput = ---------------------  * reward
                    d input
```

where the `reward` is what is provided by a Reward criterion like
VRClassReward via the reinforce method.
The criterion will normally be responsible for the following formula :

```
reward = a*(R - b)
```

where `a` is the alpha of the original paper, i.e. a reward scale,
`R` is the raw reward (usually 0 or 1), and `b` is the baseline reward,
which is often taken to be the expected raw reward `R`.

The `output` is usually sampled from a probability distribution `f()`
parameterized by the `input`.
See ReinforceBernoulli for a concrete derivation.

Also, as you can see, the gradOutput is ignored. So within a backpropagation graph,
the `Reinforce` modules will replace the backpropagated gradients (`gradOutput`)
with their own obtained from the broadcasted `reward`.

ReinforceBernoulli

Ref A. Simple Statistical Gradient-Following Algorithms for
Connectionist Reinforcement Learning

```
module = nn.ReinforceBernoulli([stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A p.230-236) for the Bernoulli probability distribution.
Inputs are bernoulli probabilities `p`.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false`, outputs are the same as the inputs.

Uses the REINFORCE algorithm (ref. A p.230-236) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : bernoulli probability mass function
- `x` : the sampled values (0 or 1) (i.e. `self.output`)
- `p` : probability of sampling a 1

the derivative of the log bernoulli w.r.t. probability `p` is :

```
d ln(f(output,input))   d ln(f(x,p))    (x - p)
---------------------- = ------------ = ---------
      d input                d p         p(1 - p)
```

ReinforceNormal

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceNormal(stdev, [stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A p.238-239) for a Normal (i.e. Gaussian) probability distribution.
Inputs are the means of the normal distribution.
The `stdev` argument specifies the standard deviation of the distribution.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false` , outputs are the same as the inputs, i.e. the means.
Uses the REINFORCE algorithm (ref. A p.238-239) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : normal probability density function
- `x` : the sampled values (i.e. `self.output`)
- `u` : mean (`input`)
- `s` : standard deviation (`self.stdev`)

the derivative of log normal w.r.t. mean `u` is :

```
d ln(f(x,u,s))   (x - u)
-------------- = -------
     d u           s^2
```

As an example, it is used to sample locations for the RecurrentAttention
module (see this example).

ReinforceGamma

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceGamma(scale, [stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A) for a Gamma probability distribution
parametrized by shape (k) and scale (theta) variables.
Inputs are the shapes of the gamma distribution.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false`, outputs are equal to the mean, defined as the product of
shape and scale ie. `k*theta`.
Uses the REINFORCE algorithm (ref. A) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : gamma probability density function
- `g` : digamma function
- `x` : the sampled values (i.e. `self.output`)
- `k` : shape (`input`)
- `t` : scale

the derivative of log gamma w.r.t. shape `k` is :

```
d ln(f(x,k,t))
```

```
-------------- = ln(x) - g(k) - ln(t)
      d k
```

ReinforceCategorical

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceCategorical([stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A) for a Categorical (i.e. Multinomial with one sample) probability distribution.
Inputs are the categorical probabilities of the distribution : `p[1], p[2], ..., p[k]`.
These are usually the output of a SoftMax.
For `n` categories, both the `input` and `output` ares of size `batchSize x n`.
During training, outputs are samples drawn from this distribution.
The outputs are returned in one-hot encoding i.e.
the output for each example has exactly one category having a 1, while the remainder are zero.
During evaluation, when `stochastic=false`, outputs are the same as the inputs, i.e. the probabilities `p`.
Uses the REINFORCE algorithm (ref. A) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : categorical probability mass function
- `x` : the sampled indices (one per sample) (`self.output` is the one-hot encoding of these indices)
- `p` : probability vector (`p[1], p[2], ..., p[k]`) (`input`)

the derivative of log categorical w.r.t. probability vector `p` is :

```
d ln(f(x,p))     1/p[i]    if i = x
------------ =
    d p          0         otherwise
```

VRClassReward

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

This Reward criterion implements the REINFORCE algoritm (ref. A) for classification models. Specifically, it is a Variance Reduces (VR) classification reinforcement leanring (reward-based) criterion.

```
vcr = nn.VRClassReward(module [, scale, criterion])
```

While it conforms to the Criterion interface (which it inherits),
it does not backpropagate gradients (except for the baseline `b`; see below).
Instead, a `reward` is broadcast to the `module` via the reinforce method.

The criterion implements the following formula :

```
reward = a*(R - b)
```

where `a` is the alpha described in Ref. A, i.e. a reward `scale` (defaults to 1),
`R` is the raw reward (0 for incorrect and 1 for correct classification),
and `b` is the baseline reward, which is often taken to be the expected raw reward `R`.

The `target` of the criterion is a tensor of class indices.
The `input` to the criterion is a table `{y,b}` where `y` is the probability
(or log-probability) of classes (usually the output of a SoftMax),
and `b` is the baseline reward discussed above.

For each example, if `argmax(y)` is equal to the `target` class, the raw reward `R = 1`,
otherwize `R = 0`.

As for `b`, its `gradInputs` are obtained from the `criterion`, which defaults to `MSECriterion`.
The `criterion`'s target is the commensurate raw reward `R`.
Using `a*(R-b)` instead of `a*R` to obtain a `reward` is what makes this class variance reduced (VR).
By reducing the variance, the training can converge faster (Ref. A).
The predicted `b` can be nothing more than the expectation `E(R)`.

Note : for RNNs with R = 1 for last step in sequence, encapsulate it

in `nn.ModuleCriterion(VRClassReward, nn.SelectTable(-1))`.

For an example, this criterion is used along with the RecurrentAttention
module to train a recurrent model for visual attention.

BinaryClassReward

```
bcr = nn.BinaryClassReward(module [, scale, criterion])
```

This module implements VRClassReward for binary classification problems.
So basically, the `input` is still a table of two tensors.
The first input tensor is of size `batchsize` containing Bernoulli probabilities.
The second input tensor is the baseline prediction described in `VRClassReward`.
The targets contain zeros and ones.

BinaryLogisticRegression

Ref A. Learning to Segment Object Candidates
This criterion implements the score criterion mentioned in (ref. A).

```
criterion = nn.BinaryLogisticRegression()
```

BinaryLogisticRegression implements following cost function for binary classification.

```
log( 1 + exp( -y_k * score(x_k) ) )
```

where `y_k` is binary target `score(x_k)` is the corresponding prediction. `y_k` has value `{-1, +1}` and `score(x_k)` has value in `[-1, +1]`.

SpatialBinaryLogisticRegression

Ref A. Learning to Segment Object Candidates

This criterion implements the spatial component of the criterion mentioned in (ref. A).

```
criterion = nn.SpatialBinaryLogisticRegression()
```

SpatialBinaryLogisticRegression implements following cost function for binary pixel classification.

```
   1
_______ sum_ij [ log( 1 + exp( -m_ij * f_ij ) ) ]
 2*w*h
```

where `m_ij` is target binary image and `f_ij` is the corresponding prediction. `m_ij` has value `{-1, +1}` and `f_ij` has value in `[-1, +1]`.