CNN wrappers

At its core, MatConvNet consists of a number of MATLAB functions (../functions/#blocks) implementing CNN building blocks. These are usually combined into complete CNNs by using one of the two CNN wrappers. The first wrapper is SimpleNN, most of which is implemented by the MATLAB function `vl_simplenn` (../mfiles/simplenn/vl_simplenn/). SimpleNN is suitable for networks that have a linear topology, i.e. a chain of computational blocks. The second wrapper is DagNN, which is implemented as the MATLAB class `dagnn.DagNN` (../mfiles/+dagnn/@DagNN/DagNN/).

SimpleNN wrapper

The SimpleNN wrapper is implemented by the function `vl_simplenn` (../mfiles/simplenn/vl_simplenn/) and a few others (../functions/#simplenn). This is a lightweight wrapper, suitable for CNN consistting of a simple chain of blocks.

To start with SimpleNN, create a `net` structure, populating the cellarray `net.layers` with a list of layers. For example:

```
net.layers{1} = struct(...
    'name', 'conv1', ...
    'type', 'conv', ...
    'weights', {{randn(10,10,3,2,'single'), randn(2,1,'singl
e')}}, ...
    'pad', 0, ...
    'stride', 1) ;
net.layers{2} = struct(...
    'name', 'relu1', ...
    'type', 'relu') ;
```

Now the convolutional and ReLU layers will be executed in sequence. The convolution has a bank of two 10x10x3 filters. Evaluation can be obtained as follows:

```
data = randn(300, 500, 3, 5, 'single') ;
res = vl_simplenn(net, data) ;
```

The structure `res` contains the result of the computation, with one entry for each variable in the architecture:

```
>> res
res =
1x3 struct array with fields:

    x
    dzdx
    dzdw
    aux
    time
    backwardTime
```

Here `x` is the variable value, `dzdx` the derivative of the CNN with respect to `x`, `dzdw` the derivative of the CNN with respect to each of the block parameters, `aux` space for custom information (e.g. the mask in dropout layers), and `time` and `backwardTime` the time spent in the forward and backward pass.

For example, `res(1).x` is the input of the CNN and `res(3).x` its output.

The derivative of the CNN can be computed as follows:

```
res = vl_simplenn(res, data, dzdy)
```

This performs both a forward and a backward pass. `dzdy` is a projection applied to the output value of the CNN (see the PDF manual (../../matconvnet-manual.pdf) to clarify this point). During training, CNNs are often terminated by a block that computes a single scalar loss value (i.e. `res(end).x` is a scalar). In this case, one often picks `dzdy = 1`.

DagNN wrapper

The DagNN wrapper is implemented by the class `daggn.DagNN` (../mfiles/+dagnn/@DagNN/DagNN/).

Creating a DagNN

A DagNN is an object of class `dagnn.DagNN` (../mfiles/+dagnn/@DagNN/DagNN/). A DAG can be created directly as follows:

```
run <MATCONVNETROOT>/matlab/vl_setupnn.m ; % activate MatConvNet
if needed
net = dagnn.DagNN() ;
```

The object is a MATLAB handle, meaning that it is passed by reference rather than by value:

```
net2 = net ; % both net and net2 refer to the same object
```

This significantly simplifies the syntax of most operations.

DagNN has a bipartite directed acyclic graph structure, where *layers* are connected to *variables* and vice-versa. Each layer receives zero or more variables and zero or more *parameters* as input and produces zero or more variables as outputs. Layers are added using the `addLayer()` method of the DagNN object. For example, the following command adds a layer an input `x1`, an output `x2`, and two parameters `filters` and `biases`.

```
convBlock = dagnn.Conv('size', [3 3 256 16], 'hasBias', true) ;
net.addLayer('conv1', convBlock, {'x1'}, {'x2'}, {'filters', 'bia
ses'}) ;
```

Next, we add a ReLU layer on top of the convolutional one:

```
reluBlock = dagnn.ReLU() ;
net.addLayer('relu1', reluBlock, {'x2'}, {'x3'}, {}) ;
```

Note that ReLU does not take any parameter. In general, blocks may have an arbitrary numbers of inputs and outputs (compatibly with the block type).

At this point, `net` contains the two blocks as well as the three variables and the two parameters. These are stored as entries in `net.layers`, `net.vars` and `net.params` respectively. For example

```
>> net.layers(1)
ans =
             name: 'conv1'
           inputs: {'x1'}
          outputs: {'x2'}
           params: {'filters'  'biases'}
     inputIndexes: 1
    outputIndexes: 2
     paramIndexes: [1 2]
            block: [1x1 dagnn.Conv]
```

contains the first block. It includes the names of the input and output variables and of the parameters (e.g. `net.layers(1).inputs`). It also contains the variable and parameter indexes for faster access (these are managed automatically by the DAG methods). Blocks are identified by name (`net.layers(1).name`) and some functions such as `removeLayer()` refer to them using these identifiers (it is an error to assign the same name to two layers). Finally, the actual layer parameters are contained in `net.layers(1).block`, which in this case is an object of type `dagnn.Conv`. The latter is, in turn, a wrapper of the `vl_nnconv` command.

The other important data members store variables and parameters. For example:

```
>> net.vars(1)
ans =
         name: 'x1'
        value: []
          der: []
        fanin: 0
       fanout: 1
     precious: 0

>> net.params(1)
ans =
         name: 'filters'
        value: []
          der: []
       fanout: 1
 learningRate: 1
  weightDecay: 1
```

Note that each variable and parameter has a `value` and a `der`ivative members. The `fanin` and `fanout` members indicated how many network layers have that particular variable/parameter as output and input, respectively. Parameters do not have a `fanin` as they are not the result of a network calculation. Variables can have `fanin` equal to zero, denoting a network input, or one. Network outputs can be identified as variables with null `fanout`.

Variables and parameters can feed into one or more layers, which results in a `fanout` equal or greater than one. For variables, fanout greater than one denotes a branching point in the DAG. For parameters, fanout greater than one allows sharing parameters between layers.

Loading and saving DagNN objects

While it is possible to save a DagNN object using MATLAB `save` command directly, this is not recommended. Instead, for compatibility and portability, as well as to save significant disk space, it is preferable to convert the object into a structure and then save that instead:

```
netStruct = net.saveobj() ;
save('myfile.mat', '-struct', 'netStruct') ;
clear netStruct ;
```

The operation can be inverted using the `loadobj()` static method of `dagnn.DagNN`:

```
netStruct = load('myfile.mat') ;
net = dagnn.DagNN.loadobj(netStruct) ;
clear netStruct ;
```

Note that in this manner the *transient* state of the object is lost. This includes the values of the variables, but not the values of the parameters, or the network structure.

Using the DagNN to evaluate the network and its derivative

So far, both parameters and variables in the DagNN object have empty values and derivatives. The command:

```
net.initParams() ;
```

can be used to initialise the model parameters to random values. For example `net.params(1).value` should now contain a 3x3x256x16 array, matching the filter dimensions specified in the example above.

The `eval()` method can now be used to evaluate the network. For example:

```
input = randn(10,15,256,1,'single') ;
net.eval({'x1', input}) ;
```

evaluates the network on a random input array. Since in general the network can have several inputs, one must specify each input as a pair `'variableName',variableValue` in a cell array.

After `eval()` completes, the `value` fields of the leaf variables in the network contain the network outputs. In this example, the single output can be recovered as follows:

```
i = net.getVarIndex('x3') ;
output = net.vars(i).value ;
```

The `getVarIndex()` method is used to obtain the variable index given its name (variables do not move around unless the network structure is changed or reloaded from disk, so that indexes can be cached).

The (projected) derivative of the CNN with respect to variables and parameters is obtained by passing, along with the input, a projection vector for each of the output variables:

```
dzdy = randn(size(output), 'single') ; % projection vector
net.eval({'x1',input},{'x3',dzdy}) ;
```

The derivatives are now stored in the corresponding `params` and `vars` structure entries. For example, the derivative with respect to the `filters` parameter can be accessed as follows:

```
p = net.getParamIndex('filters') ;
dzdfilters = net.vars(p).der ;
```

Remark: empty values of variables. Note that most intermediate variable values and derivatives are aggressively discarded during the computation in order to conserve memory. Set `net.conserveMemory` to `false` to prevent this from happening, or make individual variables precious (`net.vars(i).precious = true`).