MGL MANUAL

Contents

[in package MGL]

- **[system]** **"mgl"**

```
- _Version:_ 0.1.0
- _Description:_ `mgl` is a machine learning library for backpropagation
    neural networks, boltzmann machines, gaussian processes and more.
- _Licence:_ MIT, see COPYING.
- _Author:_ Gábor Melis <mega@retes.hu>
- _Mailto:_ [mega@retes.hu](mailto:mega@retes.hu)
- _Homepage:_ <http://melisgl.github.io/mgl>
- _Bug tracker:_ <https://github.com/melisgl/mgl/issues>
- _Source control:_ [GIT](https://github.com/melisgl/mgl.git)
- *Depends on:* alexandria, array-operations, cl-reexport, closer-mop, lla,
↪  mgl-gnuplot, [mgl-mat][caca], [mgl-pax][6fdb], [named-readtables][718a],
↪  num-utils, pythonic-string-reader, swank(?)
```

1 Introduction

1.1 Overview

MGL is a Common Lisp machine learning library by Gábor Melis with some parts originally contributed by Ravenpack International. It mainly concentrates on various forms of neural networks (boltzmann machines, feed-forward and recurrent backprop nets). Most of MGL is built on top of MGL-MAT so it has BLAS and CUDA support.

In general, the focus is on power and performance not on ease of use. Perhaps one day there will be a cookie cutter interface with restricted functionality if a reasonable compromise is found between power and utility.

1.2 Links

Here is the official repository and the HTML documentation for the latest version.

1.3 Dependencies

MGL used to rely on LLA to interface to BLAS and LAPACK. That's mostly history by now, but configuration of foreign libraries is still done via LLA. See the README in LLA on how to set things up. Note that these days OpenBLAS is easier to set up and just as fast as ATLAS.

CL-CUDA and MGL-MAT are the two main dependencies and also the ones not yet in quicklisp, so just drop them into `quicklisp/local-projects/`. If there is no suitable GPU on the system

or the CUDA SDK is not installed, MGL will simply fall back on using BLAS and Lisp code. Wrapping code in `mgl-mat:with-cuda*` is basically all that's needed to run on the GPU, and with `mgl-mat:cuda-available-p` one can check whether the GPU is really being used.

1.4 Code Organization

MGL consists of several packages dedicated to different tasks. For example, package `mgl-resample` is about Resampling and `mgl-gd` is about Gradient Descent and so on. On one hand, having many packages makes it easier to cleanly separate API and implementation and also to explore into a specific task. At other times, they can be a hassle, so the `mgl` package itself reexports every external symbol found in all the other packages that make up MGL and MGL-MAT (see MAT Manual) on which it heavily relies.

One exception to this rule is the bundled, but independent MGL-GNUPLOT library.

The built in tests can be run with:

```
(ASDF:OOS 'ASDF:TEST-OP '#:MGL)
```

Note, that most of the tests are rather stochastic and can fail once in a while.

1.5 Glossary

Ultimately machine learning is about creating **models** of some domain. The observations in the modelled domain are called **instances** (also known as examples or samples). Sets of instances are called **datasets**. Datasets are used when fitting a model or when making **predictions**. Sometimes the word predictions is too specific, and the results obtained from applying a model to some instances are simply called **results**.

2 Common Stuff

[in package MGL-COMMON]

- **[generic-function]** **name** *object*

    - **[function]** **name=** *x y*

      Return `t` if X and Y are `eql(0 1)` or if they are structured components whose elements are `equal`. Strings and bit-vectors are `equal` if they are the same length and have identical components. Other arrays must be `eq` to be `equal`.

    - **[generic-function]** **size** *object*

    - **[generic-function]** **nodes** *object*

      Returns a `mgl-mat:mat` object representing the state or result of `object`. The first dimension of the returned matrix is equal to the number of stripes.

    - **[generic-function]** **default-value** *object*

    - **[generic-function]** **group-size** *object*

- **[generic-function]** **batch-size** *object*
- **[generic-function]** **weights** *object*
- **[generic-function]** **scale** *object*

3 Datasets

[in package MGL-DATASET]

An instance can often be any kind of object of the user's choice. It is typically represented by a set of numbers which is called a feature vector or by a structure holding the feature vector, the label, etc. A dataset is a `sequence` of such instances or a Samplers object that produces instances.

- **[function]** **map-dataset** *fn dataset*

  Call `fn` with each instance in `dataset`. This is basically equivalent to iterating over the elements of a sequence or a sampler (see Samplers).

- **[function]** **map-datasets** *fn datasets &key (impute nil imputep)*

  Call `fn` with a list of instances, one from each dataset in `datasets`. Return nothing. If `impute` is specified then iterate until the largest dataset is consumed imputing `impute` for missing values. If `impute` is not specified then iterate until the smallest dataset runs out.

```
(map-datasets #'prin1 '((0 1 2) (:a :b)))
.. (0 :A)(1 :B)

(map-datasets #'prin1 '((0 1 2) (:a :b)) :impute nil)
.. (0 :A)(1 :B)(2 NIL)
```

  It is of course allowed to mix sequences with samplers:

```
(map-datasets #'prin1
              (list '(0 1 2)
                    (make-sequence-sampler '(:a :b) :max-n-samples 2)))
.. (0 :A)(1 :B)
```

3.1 Samplers

Some algorithms do not need random access to the entire dataset and can work with a stream observations. Samplers are simple generators providing two functions: `sample` and `finishedp`.

- **[generic-function]** **sample** *sampler*

  If `sampler` has not run out of data (see `finishedp`) `sample` returns an object that represents a sample from the world to be experienced or, in other words, simply something the can be used as input for training or prediction. It is not allowed to call `sample` if `sampler` is `finishedp`.

- **[generic-function]** **finishedp** *sampler*

See if `sampler` has run out of examples.

- **[function]** **list-samples** *sampler max-size*

  Return a list of samples of length at most `max-size` or less if `sampler` runs out.

- **[function]** **make-sequence-sampler** *seq &key max-n-samples*

  Create a sampler that returns elements of `seq` in their original order. If `max-n-samples` is non-nil, then at most `max-n-samples` are sampled.

- **[function]** **make-random-sampler** *seq &key max-n-samples (reorder #'mgl-resample:shuffle)*

  Create a sampler that returns elements of `seq` in random order. If `max-n-samples` is non-nil, then at most `max-n-samples` are sampled. The first pass over a shuffled copy of `seq`, and this copy is reshuffled whenever the sampler reaches the end of it. Shuffling is performed by calling the `reorder` function.

- **[variable]** ***infinitely-empty-dataset*** *#<function-sampler "infinitely empty" >*

  This is the default dataset for `mgl-opt:minimize`. It's an infinite stream of `nil`s.

3.1.1 Function Sampler

- **[class]** **function-sampler**

  A sampler with a function in its `generator` that produces a stream of samples which may or may not be finite depending on `max-n-samples`. `finishedp` returns `t` iff `max-n-samples` is non-nil, and it's not greater than the number of samples generated (`n-samples`).

```
(list-samples (make-instance 'function-sampler
                             :generator (lambda ()
                                          (random 10))
                             :max-n-samples 5)
              10)
=> (3 5 2 3 3)
```

- **[reader]** **generator** *function-sampler (:generator)*

  A generator function of no arguments that returns the next sample.

- **[accessor]** **max-n-samples** *function-sampler (:max-n-samples = nil)*

- **[reader]** **name** *function-sampler (:name = nil)*

  An arbitrary object naming the sampler. Only used for printing the sampler object.

- **[reader]** **n-samples** *function-sampler (:n-samples = 0)*

4 Resampling

[in package MGL-RESAMPLE]

The focus of this package is on resampling methods such as cross-validation and bagging which can be used for model evaluation, model selection, and also as a simple form of ensembling. Data partitioning and sampling functions are also provided because they tend to be used together with resampling.

4.1 Shuffling

- **[function]** **shuffle** *seq*

  Copy of `seq` and shuffle it using Fisher-Yates algorithm.

- **[function]** **shuffle!** *seq*

  Shuffle `seq` using Fisher-Yates algorithm.

4.2 Partitions

The following functions partition a dataset (currently only `sequence`s are supported) into a number of partitions. For each element in the original dataset there is exactly one partition that contains it.

- **[function]** **fracture** *fractions seq &key weight*

  Partition `seq` into a number of subsequences. `fractions` is either a positive integer or a list of non-negative real numbers. `weight` is `nil` or a function that returns a non-negative real number when called with an element from `seq`. If `fractions` is a positive integer then return a list of that many subsequences with equal sum of weights bar rounding errors, else partition `seq` into subsequences, where the sum of weights of subsequence I is proportional to element I of `fractions`. If `weight` is `nil`, then it's element is assumed to have the same weight.

  To split into 5 sequences:

  ```
  (fracture 5 '(0 1 2 3 4 5 6 7 8 9))
  => ((0 1) (2 3) (4 5) (6 7) (8 9))
  ```

  To split into two sequences whose lengths are proportional to 2 and 3:

  ```
  (fracture '(2 3) '(0 1 2 3 4 5 6 7 8 9))
  => ((0 1 2 3) (4 5 6 7 8 9))
  ```

- **[function]** **stratify** *seq &key (key #'identity) (test #'eql)*

  Return the list of strata of `seq`. `seq` is a sequence of elements for which the function `key` returns the class they belong to. Such classes are opaque objects compared for equality with `test`. A stratum is a sequence of elements with the same (under `test`) `key`.

```
(stratify '(0 1 2 3 4 5 6 7 8 9) :key #'evenp)
=> ((0 2 4 6 8) (1 3 5 7 9))
```

- **[function]** **fracture-stratified** *fractions seq &key (key #'identity) (test #'eql) weight*

  Similar to `fracture`, but also makes sure that keys are evenly distributed among the partitions (see `stratify`). It can be useful for classification tasks to partition the data set while keeping the distribution of classes the same.

  Note that the sets returned are not in random order. In fact, they are sorted internally by `key`.

  For example, to make two splits with approximately the same number of even and odd numbers:

```
(fracture-stratified 2 '(0 1 2 3 4 5 6 7 8 9) :key #'evenp)
=> ((0 2 1 3) (4 6 8 5 7 9))
```

4.3 Cross-validation

- **[function]** **cross-validate** *data fn &key (n-folds 5) (folds (alexandria:iota n-folds)) (split-fn #'split-fold/mod) pass-fold*

  Map `fn` over the `folds` of `data` split with `split-fn` and collect the results in a list. The simplest demonstration is:

```
(cross-validate '(0 1 2 3 4)
                (lambda (test training)
                 (list test training))
                :n-folds 5)
=> (((0) (1 2 3 4))
    ((1) (0 2 3 4))
    ((2) (0 1 3 4))
    ((3) (0 1 2 4))
    ((4) (0 1 2 3)))
```

  Of course, in practice one would typically train a model and return the trained model and/or its score on `test`. Also, sometimes one may want to do only some of the folds and remember which ones they were:

```
(cross-validate '(0 1 2 3 4)
                (lambda (fold test training)
                 (list :fold fold test training))
                :folds '(2 3)
                :pass-fold t)
=> ((:fold 2 (2) (0 1 3 4))
    (:fold 3 (3) (0 1 2 4)))
```

  Finally, the way the data is split can be customized. By default `split-fold/mod` is called with the arguments `data`, the fold (from among `folds`) and `n-folds`. `split-fold/mod` returns two values which are then passed on to `fn`. One can use `split-fold/cont` or `split-stratified` or any other function that works with these arguments. The only real

constraint is that `fn` has to take as many arguments (plus the fold argument if `pass-fold`) as `split-fn` returns.

- **[function]** **split-fold/mod** *seq fold n-folds*

  Partition `seq` into two sequences: one with elements of `seq` with indices whose remainder is `fold` when divided with `n-folds`, and a second one with the rest. The second one is the larger set. The order of elements remains stable. This function is suitable as the `split-fn` argument of `cross-validate`.

- **[function]** **split-fold/cont** *seq fold n-folds*

  Imagine dividing `seq` into `n-folds` subsequences of the same size (bar rounding). Return the subsequence of index `fold` as the first value and the all the other subsequences concatenated into one as the second value. The order of elements remains stable. This function is suitable as the `split-fn` argument of `cross-validate`.

- **[function]** **split-stratified** *seq fold n-folds &key (key #'identity) (test #'eql) weight*

  Split `seq` into `n-folds` partitions (as in `fracture-stratified`). Return the partition of index `fold` as the first value, and the concatenation of the rest as the second value. This function is suitable as the `split-fn` argument of `cross-validate` (mostly likely as a closure with `key`, `test`, `weight` bound).

4.4 Bagging

- **[function]** **bag** *seq fn &key (ratio 1) n weight (replacement t) key (test #'eql) (random-state *random-state*)*

  Sample from `seq` with `sample-from` (passing `ratio`, `weight`, `replacement`), or `sample-stratified` if `key` is not `nil`. Call `fn` with the sample. If `n` is `nil` then keep repeating this until `fn` performs a non-local exit. Else `n` must be a non-negative integer, `n` iterations will be performed, the primary values returned by `fn` collected into a list and returned. See `sample-from` and `sample-stratified` for examples.

- **[function]** **sample-from** *ratio seq &key weight replacement (random-state *random-state*)*

  Return a sequence constructed by sampling with or without `replacement` from `seq`. The sum of weights in the result sequence will approximately be the sum of weights of `seq` times `ratio`. If `weight` is `nil` then elements are assumed to have equal weights, else `weight` should return a non-negative real number when called with an element of `seq`.

  To randomly select half of the elements:

```
(sample-from 1/2 '(0 1 2 3 4 5))
=> (5 3 2)
```

  To randomly select some elements such that the sum of their weights constitute about half of the sum of weights across the whole sequence:

```
(sample-from 1/2 '(0 1 2 3 4 5 6 7 8 9) :weight #'identity)
=> ;; sums to 28 that's near 45/2
   (9 4 1 6 8)
```

To sample with replacement (that is, allowing the element to be sampled multiple times):

```
(sample-from 1 '(0 1 2 3 4 5) :replacement t)
=> (1 1 5 1 4 4)
```

- **[function]** **sample-stratified** *ratio seq &key weight replacement (key #'identity) (test #'eql) (random-state *random-state*)*

  Like `sample-from` but makes sure that the weighted proportion of classes in the result is approximately the same as the proportion in `seq`. See `stratify` for the description of `key` and `test`.

4.5 CV Bagging

- **[function]** **bag-cv** *data fn &key n (n-folds 5) (folds (alexandria:iota n-folds)) (split-fn #'split-fold/mod) pass-fold (random-state *random-state*)*

  Perform cross-validation on different shuffles of `data` `n` times and collect the results. Since `cross-validate` collects the return values of `fn`, the return value of this function is a list of lists of `fn` results. If `n` is `nil`, don't collect anything just keep doing repeated CVs until `fn` performs a non-local exit.

  The following example simply collects the test and training sets for 2-fold CV repeated 3 times with shuffled data:

```
;;; This is non-deterministic.
(bag-cv '(0 1 2 3 4) #'list :n 3 :n-folds 2)
=> ((((2 3 4) (1 0))
     ((1 0) (2 3 4)))
    (((2 1 0) (4 3))
     ((4 3) (2 1 0)))
    (((1 0 3) (2 4))
     ((2 4) (1 0 3))))
```

  CV bagging is useful when a single CV is not producing stable results. As an ensemble method, CV bagging has the advantage over bagging that each example will occur the same number of times and after the first CV is complete there is a complete but less reliable estimate for each example which gets refined by further CVs.

4.6 Miscellaneous Operations

- **[function]** **spread-strata** *seq &key (key #'identity) (test #'eql)*

  Return a sequence that's a reordering of `seq` such that elements belonging to different strata (under `key` and `test`, see `stratify`) are distributed evenly. The order of elements belonging to the same stratum is unchanged.

For example, to make sure that even and odd numbers are distributed evenly:

```
(spread-strata '(0 2 4 6 8 1 3 5 7 9) :key #'evenp)
=> (0 1 2 3 4 5 6 7 8 9)
```

Same thing with unbalanced classes:

```
(spread-strata (vector 0 2 3 5 6 1 4)
               :key (lambda (x)
                      (if (member x '(1 4))
                          t
                          nil)))
=> #(0 1 2 3 4 5 6)
```

- **[function]** **zip-evenly** *seqs &key result-type*

  Make a single sequence out of the sequences in `seqs` so that in the returned sequence indices of elements belonging to the same source sequence are spread evenly across the whole range. The result is a list is `result-type` is `list`(`0` `1`), it's a vector if `result-type` is `vector`(`0` `1`). If `result-type` is `nil`, then it's determined by the type of the first sequence in `seqs`.

```
(zip-evenly '((0 2 4) (1 3)))
=> (0 1 2 3 4)
```

5 Core

[in package MGL-CORE]

5.1 Persistence

- **[function]** **load-state** *filename object*

  Load weights of `object` from `filename`. Return `object`.

- **[function]** **save-state** *filename object &key (if-exists :error) (ensure t)*

  Save weights of `object` to `filename`. If `ensure`, then `ensure-directories-exist` is called on `filename`. `if-exists` is passed on to `open`. Return `object`.

- **[function]** **read-state** *object stream*

  Read the weights of `object` from the bivalent `stream` where weights mean the learnt parameters. There is currently no sanity checking of data which will most certainly change in the future together with the serialization format. Return `object`.

- **[function]** **write-state** *object stream*

  Write weight of `object` to the bivalent `stream`. Return `object`.

- **[generic-function]** **read-state*** *object stream context*

  This is the extension point for `read-state`. It is guaranteed that primary `read-state*` methods will be called only once for each `object` (under `eq`). `context` is an opaque object

and must be passed on to any recursive `read-state*` calls.

- **[generic-function]** **write-state*** *object stream context*

  This is the extension point for `write-state`. It is guaranteed that primary `write-state*` methods will be called only once for each `object` (under `eq`). `context` is an opaque object and must be passed on to any recursive `write-state*` calls.

5.2 Batch Processing

Processing instances one by one during training or prediction can be slow. The models that support batch processing for greater efficiency are said to be *striped*.

Typically, during or after creating a model, one sets `max-n-stripes` on it a positive integer. When a batch of instances is to be fed to the model it is first broken into subbatches of length that's at most `max-n-stripes`. For each subbatch, `set-input` (FIXDOC) is called and a before method takes care of setting `n-stripes` to the actual number of instances in the subbatch. When `max-n-stripes` is set internal data structures may be resized which is an expensive operation. Setting `n-stripes` is a comparatively cheap operation, often implemented as matrix reshaping.

Note that for models made of different parts (for example, `mgl-bp:bpn` consists of `mgl-bp:lump`s) , setting these values affects the constituent parts, but one should never change the number stripes of the parts directly because that would lead to an internal inconsistency in the model.

- **[generic-function]** **max-n-stripes** *object*

  The number of stripes with which the `object` is capable of dealing simultaneously.

- **[generic-function]** **set-max-n-stripes** *max-n-stripes object*

  Allocate the necessary stuff to allow for `max-n-stripes` number of stripes to be worked with simultaneously in `object`. This is called when `max-n-stripes` is `setf`'ed.

- **[generic-function]** **n-stripes** *object*

  The number of stripes currently present in `object`. This is at most `max-n-stripes`.

- **[generic-function]** **set-n-stripes** *n-stripes object*

  Set the number of stripes (out of `max-n-stripes`) that are in use in `object`. This is called when `n-stripes` is `setf`'ed.

- **[macro]** **with-stripes** *specs &body body*

  Bind start and optionally end indices belonging to stripes in striped objects.

```
(WITH-STRIPES ((STRIPE1 OBJECT1 START1 END1)
               (STRIPE2 OBJECT2 START2)
               ...)
 ...)
```

  This is how one's supposed to find the index range corresponding to the Nth input in an input lump of a bpn:

```
(with-stripes ((n input-lump start end))
  (loop for i upfrom start below end
        do (setf (mref (nodes input-lump) i) 0d0)))
```

Note how the input lump is striped, but the matrix into which we are indexing (`nodes`) is not known to `with-stripes`. In fact, for lumps the same stripe indices work with `nodes` and `mgl-bp:derivatives`.

- **[generic-function]** **stripe-start** *stripe object*

  Return the start index of `stripe` in some array or matrix of `object`.

- **[generic-function]** **stripe-end** *stripe object*

  Return the end index (exclusive) of `stripe` in some array or matrix of `object`.

- **[generic-function]** **set-input** *instances model*

  Set `instances` as inputs in `model`. `instances` is always a `sequence` of instances even for models not capable of batch operation. It sets `n-stripes` to (`length` `instances`) in a `:before` method.

- **[function]** **map-batches-for-model** *fn dataset model*

  Call `fn` with batches of instances from `dataset` suitable for `model`. The number of instances in a batch is `max-n-stripes` of `model` or less if there are no more instances left.

- **[macro]** **do-batches-for-model** *(batch (dataset model)) &body body*

  Convenience macro over `map-batches-for-model`.

5.3 Executors

- **[generic-function]** **map-over-executors** *fn instances prototype-executor*

  Divide `instances` between executors that perform the same function as `prototype-executor` and call `fn` with the instances and the executor for which the instances are.

  Some objects conflate function and call: the forward pass of a `mgl-bp:bpn` computes output from inputs so it is like a function but it also doubles as a function call in the sense that the bpn (function) object changes state during the computation of the output. Hence not even the forward pass of a bpn is thread safe. There is also the restriction that all inputs must be of the same size.

  For example, if we have a function that builds bpn a for an input of a certain size, then we can create a factory that creates bpns for a particular call. The factory probably wants to keep the weights the same though. In Parameterized Executor Cache, `make-executor-with-parameters` is this factory.

  Parallelization of execution is another possibility `map-over-executors` allows, but there is no prebuilt solution for it, yet.

  The default implementation simply calls `fn` with `instances` and `prototype-executor`.

- **[macro]** **do-executors** *(instances object) &body body*

  Convenience macro on top of `map-over-executors`.

5.3.1 Parameterized Executor Cache

- **[class]** **parameterized-executor-cache-mixin**

  Mix this into a model, implement `instance-to-executor-parameters` and `make-executor-with-parameters` and `do-executors` will be to able build executors suitable for different instances. The canonical example is using a BPN to compute the means and convariances of a gaussian process. Since each instance is made of a variable number of observations, the size of the input is not constant, thus we have a bpn (an executor) for each input dimension (the parameters).

- **[generic-function]** **make-executor-with-parameters** *parameters cache*

  Create a new executor for `parameters`. `cache` is a `parameterized-executor-cache-mixin`. In the BPN gaussian process example, `parameters` would be a list of input dimensions.

- **[generic-function]** **instance-to-executor-parameters** *instance cache*

  Return the parameters for an executor able to handle `instance`. Called by `map-over-executors` on `cache` (that's a `parameterized-executor-cache-mixin`). The returned parameters are keys in an `equal` parameters->executor hash table.

6 Monitoring

[in package MGL-CORE]

When training or applying a model, one often wants to track various statistics. For example, in the case of training a neural network with cross-entropy loss, these statistics could be the average cross-entropy loss itself, classification accuracy, or even the entire confusion matrix and sparsity levels in hidden layers. Also, there is the question of what to do with the measured values (log and forget, add to some counter or a list).

So there may be several phases of operation when we want to keep an eye on. Let's call these **events**. There can also be many fairly independent things to do in response to an event. Let's call these **monitors**. Some monitors are a composition of two operations: one that extracts some measurements and another that aggregates those measurements. Let's call these two **measurers** and **counters**, respectively.

For example, consider training a backpropagation neural network. We want to look at the state of of network just after the backward pass. `mgl-bp:bp-learner` has a `monitors` event hook corresponding to the moment after backpropagating the gradients. Suppose we are interested in how the training cost evolves:

```
(push (make-instance 'monitor
                     :measurer (lambda (instances bpn)
                                 (declare (ignore instances))
```

```
                                  (mgl-bp:cost bpn))
                     :counter (make-instance 'basic-counter))
      (monitors learner))
```

During training, this monitor will track the cost of training examples behind the scenes. If we want to print and reset this monitor periodically we can put another monitor on `mgl-opt:iterative-optimizer`'s `mgl-opt:on-n-instances-changed` accessor:

```
(push (lambda (optimizer gradient-source n-instances)
        (declare (ignore optimizer))
        (when (zerop (mod n-instances 1000))
          (format t "n-instances: ~S~%" n-instances)
          (dolist (monitor (monitors gradient-source))
            (when (counter monitor)
              (format t "~A~%" (counter monitor))
              (reset-counter (counter monitor)))))))
      (mgl-opt:on-n-instances-changed optimizer))
```

Note that the monitor we push can be anything as long as `apply-monitor` is implemented on it with the appropriate signature. Also note that the `zerop` + `mod`(`0` `1`) logic is fragile, so you will likely want to use `mgl-opt:monitor-optimization-periodically` instead of doing the above.

So that's the general idea. Concrete events are documented where they are signalled. Often there are task specific utilities that create a reasonable set of default monitors (see Classification Monitors).

- **[function]** **apply-monitors** *monitors &rest arguments*

  Call `apply-monitor` on each monitor in `monitors` and `arguments`. This is how an event is fired.

- **[generic-function]** **apply-monitor** *monitor &rest arguments*

  Apply `monitor` to `arguments`. This sound fairly generic, because it is. `monitor` can be anything, even a simple function or symbol, in which case this is just `cl:apply`. See Monitors for more.

- **[generic-function]** **counter** *monitor*

  Return an object representing the state of `monitor` or `nil`, if it doesn't have any (say because it's a simple logging function). Most monitors have counters into which they accumulate results until they are printed and reset. See Counters for more.

- **[function]** **monitor-model-results** *fn dataset model monitors*

  Call `fn` with batches of instances from `dataset` until it runs out (as in `do-batches-for-model`). `fn` is supposed to apply `model` to the batch and return some kind of result (for neural networks, the result is the model state itself). Apply `monitors` to each batch and the result returned by `fn` for that batch. Finally, return the list of counters of `monitors`.

  The purpose of this function is to collect various results and statistics (such as error measures) efficiently by applying the model only once, leaving extraction of quantities of

interest from the model's results to `monitors`.

See the model specific versions of this functions such as `mgl-bp:monitor-bpn-results`.

- **[generic-function]** **monitors** *object*

  Return monitors associated with `object`. See various methods such as `monitors` for more documentation.

6.1 Monitors

- **[class]** **monitor**

  A monitor that has another monitor called `measurer` embedded in it. When this monitor is applied, it applies the measurer and passes the returned values to `add-to-counter` called on its `counter` slot. One may further specialize `apply-monitor` to change that.

  This class is useful when the same event monitor is applied repeatedly over a period and its results must be aggregated such as when training statistics are being tracked or when predictions are begin made. Note that the monitor must be compatible with the event it handles. That is, the embedded `measurer` must be prepared to take the arguments that are documented to come with the event.

- **[reader]** **measurer** *monitor (:measurer)*

  This must be a monitor itself which only means that `apply-monitor` is defined on it (but see Monitoring). The returned values are aggregated by `counter`. See Measurers for a library of measurers.

- **[reader]** **counter** *monitor (:counter)*

  The `counter` of a monitor carries out the aggregation of results returned by `measurer`. The See Counters for a library of counters.

6.2 Measurers

`measurer` is a part of `monitor` objects, an embedded monitor that computes a specific quantity (e.g. classification accuracy) from the arguments of event it is applied to (e.g. the model results). Measurers are often implemented by combining some kind of model specific extractor with a generic measurer function.

All generic measurer functions return their results as multiple values matching the arguments of `add-to-counter` for a counter of a certain type (see Counters) so as to make them easily used in a `monitor`:

```
(multiple-value-call #'add-to-counter <some-counter>
                     <call-to-some-measurer>)
```

The counter class compatible with the measurer this way is noted for each function.

For a list of measurer functions see Classification Measurers.

6.3 Counters

- **[generic-function]** **add-to-counter** *counter &rest args*

  Add `args` to `counter` in some way. See specialized methods for type specific documentation. The kind of arguments to be supported is the what the measurer functions (see Measurers) intended to be paired with the counter return as multiple values.

- **[generic-function]** **counter-values** *counter*

  Return any number of values representing the state of `counter`. See specialized methods for type specific documentation.

- **[generic-function]** **counter-raw-values** *counter*

  Return any number of values representing the state of `counter` in such a way that passing the returned values as arguments `add-to-counter` on a fresh instance of the same type recreates the original state.

- **[generic-function]** **reset-counter** *counter*

  Restore state of `counter` to what it was just after creation.

6.3.1 Attributes

- **[class]** **attributed**

  This is a utility class that all counters subclass. The `attributes` plist can hold basically anything. Currently the attributes are only used when printing and they can be specified by the user. The monitor maker functions such as those in Classification Monitors also add attributes of their own to the counters they create.

  With the `:prepend-attributes` initarg when can easily add new attributes without clobbering the those in the `:initform`, (`:type` "rmse") in this case.

```
(princ (make-instance 'rmse-counter
                      :prepend-attributes '(:event "pred."
                                            :dataset "test")))
;; pred. test rmse: 0.000e+0 (0)
=> #<RMSE-COUNTER pred. test rmse: 0.000e+0 (0)>
```

- **[accessor]** **attributes** *attributed (:attributes = nil)*

  A plist of attribute keys and values.

- **[method]** **name** *(attributed attributed)*

  Return a string assembled from the values of the `attributes` of `attributed`. If there are multiple entries with the same key, then they are printed near together.

  Values may be padded according to an enclosing `with-padded-attribute-printing`.

- **[macro]** **with-padded-attribute-printing** *(attributeds) &body body*

Note the width of values for each attribute key which is the number of characters in the value's `princ-to-string`'ed representation. In `body`, if attributes with they same key are printed they are forced to be at least this wide. This allows for nice, table-like output:

```
(let ((attributeds
       (list (make-instance 'basic-counter
                            :attributes '(:a 1 :b 23 :c 456))
             (make-instance 'basic-counter
                            :attributes '(:a 123 :b 45 :c 6)))))
  (with-padded-attribute-printing (attributeds)
    (map nil (lambda (attributed)
               (format t "~A~%" attributed))
         attributeds)))
;; 1   23 456: 0.000e+0 (0)
;; 123 45 6  : 0.000e+0 (0)
```

- **[function]** **log-padded** *attributeds*

  Log (see `log-msg`) `attributeds` non-escaped (as in `princ` or ~A) with the output being as table-like as possible.

6.3.2 Counter classes

In addition to the really basic ones here, also see Classification Counters.

- **[class]** **basic-counter** *attributed*

  A simple counter whose `add-to-counter` takes two additional parameters: an increment to the internal sums of called the `numerator` and `denominator`. `counter-values` returns two values:

  - `numerator` divided by `denominator` (or 0 if `denominator` is 0) and
  - `denominator`

  Here is an example the compute the mean of 5 things received in two batches:

```
(let ((counter (make-instance 'basic-counter)))
  (add-to-counter counter 6.5 3)
  (add-to-counter counter 3.5 2)
  counter)
=> #<BASIC-COUNTER 2.00000e+0 (5)>
```

- **[class]** **rmse-counter** *basic-counter*

  A `basic-counter` with whose nominator accumulates the square of some statistics. It has the attribute `:type` "rmse". `counter-values` returns the square root of what `basic-counter`'s `counter-values` would return.

```
(let ((counter (make-instance 'rmse-counter)))
  (add-to-counter counter (+ (* 3 3) (* 4 4)) 2)
  counter)
=> #<RMSE-COUNTER rmse: 3.53553e+0 (2)>
```

- **[class]** **concat-counter** *attributed*

  A counter that simply concatenates sequences.

```
(let ((counter (make-instance 'concat-counter)))
  (add-to-counter counter '(1 2 3) #(4 5))
  (add-to-counter counter '(6 7))
  (counter-values counter))
=> (1 2 3 4 5 6 7)
```

- **[reader]** **concatenation-type** *concat-counter (:concatenation-type = 'list)*

  A type designator suitable as the RESULT-TYPE argument to `concatenate`.

7 Classification

[in package MGL-CORE]

To be able to measure classification related quantities, we need to define what the label of an instance is. Customization is possible by implementing a method for a specific type of instance, but these functions only ever appear as defaults that can be overridden.

- **[generic-function]** **label-index** *instance*

  Return the label of `instance` as a non-negative integer.

- **[generic-function]** **label-index-distribution** *instance*

  Return a one dimensional array of probabilities representing the distribution of labels. The probability of the label with `label-index` `i` is element at index `i` of the returned arrray.

The following two functions are basically the same as the previous two, but in batch mode: they return a sequence of label indices or distributions. These are called on results produced by models. Implement these for a model and the monitor maker functions below will automatically work. See FIXDOC: for bpn and boltzmann.

- **[generic-function]** **label-indices** *results*

  Return a sequence of label indices for `results` produced by some model for a batch of instances. This is akin to `label-index`.

- **[generic-function]** **label-index-distributions** *result*

  Return a sequence of label index distributions for `results` produced by some model for a batch of instances. This is akin to `label-index-distribution`.

7.1 Classification Monitors

The following functions return a list monitors. The monitors are for events of signature (`instances model`) such as those produced by `monitor-model-results` and its various model specific variations. They are model-agnostic functions, extensible to new classifier types.

- **[function]** **make-classification-accuracy-monitors** *model &key operation-mode attributes (label-index-fn #'label-index)*

  Return a list of `monitor` objects associated with `classification-accuracy-counter`s. `label-index-fn` is a function like `label-index`. See that function for more.

  Implemented in terms of `make-classification-accuracy-monitors*`.

- **[function]** **make-cross-entropy-monitors** *model &key operation-mode attributes (label-index-distribution-fn #'label-index-distribution)*

  Return a list of `monitor` objects associated with `cross-entropy-counter`s. `label-index-distribution-fn` is a function like `label-index-distribution`. See that function for more.

  Implemented in terms of `make-cross-entropy-monitors*`.

- **[function]** **make-label-monitors** *model &key operation-mode attributes (label-index-fn #'label-index) (label-index-distribution-fn #'label-index-distribution)*

  Return classification accuracy and cross-entropy monitors. See `make-classification-accuracy-monitors` and `make-cross-entropy-monitors` for a description of paramters.

The monitor makers above can be extended to support new classifier types via the following generic functions.

- **[generic-function]** **make-classification-accuracy-monitors*** *model operation-mode label-index-fn attributes*

  Identical to `make-classification-accuracy-monitors` bar the keywords arguments. Specialize this to add to support for new model types. The default implementation also allows for some extensibility: if `label-indices` is defined on `model`, then it will be used to extract label indices from model results.

- **[generic-function]** **make-cross-entropy-monitors*** *model operation-mode label-index-distribution-fn attributes*

  Identical to `make-cross-entropy-monitors` bar the keywords arguments. Specialize this to add to support for new model types. The default implementation also allows for some extensibility: if `label-index-distributions` is defined on `model`, then it will be used to extract label distributions from model results.

7.2 Classification Measurers

The functions here compare some known good solution (also known as *ground truth* or *target*) to a prediction or approximation and return some measure of their [dis]similarity. They are model independent, hence one has to extract the ground truths and predictions first. Rarely used directly, they are mostly hidden behind Classification Monitors.

- **[function]** **measure-classification-accuracy** *truths predictions &key (test #'eql) truth-key prediction-key weight*

Return the number of correct classifications and as the second value the number of instances (equal to length of `truths` in the non-weighted case). `truths` (keyed by `truth-key`) is a sequence of opaque class labels compared with `test` to another sequence of classes labels in `predictions` (keyed by `prediction-key`). If `weight` is non-nil, then it is a function that returns the weight of an element of `truths`. Weighted cases add their weight to both counts (returned as the first and second values) instead of 1 as in the non-weighted case.

Note how the returned values are suitable for `multiple-value-call` with #'`add-to-counter` and a `classification-accuracy-counter`.

- **[function]** **measure-cross-entropy** *truths predictions &key truth-key prediction-key (min-prediction-pr 1.0d-15)*

  Return the sum of the cross-entropy between pairs of elements with the same index of `truths` and `predictions`. `truth-key` is a function that's when applied to an element of `truths` returns a sequence representing some kind of discrete target distribution (P in the definition below). `truth-key` may be `nil` which is equivalent to the `identity` function. `prediction-key` is the same kind of key for `predictions`, but the sequence it returns represents a distribution that approximates (Q below) the true one.

  Cross-entropy of the true and approximating distributions is defined as:

```
cross-entropy(p,q) = - sum_i p(i) * log(q(i))
```

  of which this function returns the sum over the pairs of elements of `truths` and `predictions` keyed by `truth-key` and `prediction-key`.

  Due to the logarithm, if q(i) is close to zero, we run into numerical problems. To prevent this, all q(i) that are less than `min-prediction-pr` are treated as if they were `min-prediction-pr`.

  The second value returned is the sum of p(i) over all `truths` and all `i`. This is normally equal to `(length truths)`, since elements of `truths` represent a probability distribution, but this is not enforced which allows relative importance of elements to be controlled.

  The third value returned is a plist that maps each index occurring in the distribution sequences to a list of two elements:

```
 sum_j p_j(i) * log(q_j(i))
```

  and

```
sum_j p_j(i)
```

  where `j` indexes into `truths` and `predictions`.

```
(measure-cross-entropy '((0 1 0)) '((0.1 0.7 0.2)))
=> 0.35667497
   1
   (2 (0.0 0)
    1 (0.35667497 1)
    0 (0.0 0))
```

Note how the returned values are suitable for `multiple-value-call` with #'`add-to-counter` and a `cross-entropy-counter`.

- **[function]** **measure-roc-auc** *predictions pred &key (key #'identity) weight*

  Return the area under the ROC curve for `predictions` representing predictions for a binary classification problem. `pred` is a predicate function for deciding whether a prediction belongs to the so called positive class. `key` returns a number for each element which is the predictor's idea of how much that element is likely to belong to the class, although it's not necessarily a probability.

  If `weight` is `nil`, then all elements of `predictions` count as 1 towards the unnormalized sum within AUC. Else `weight` must be a function like `key`, but it should return the importance (a positive real number) of elements. If the weight of an prediction is 2 then it's as if there were another identical copy of that prediction in `predictions`.

  The algorithm is based on algorithm 2 in the paper 'An introduction to ROC analysis' by Tom Fawcett.

  ROC AUC is equal to the probability of a randomly chosen positive having higher `key` (score) than a randomly chosen negative element. With equal scores in mind, a more precise version is: AUC is the expectation of the above probability over all possible sequences sorted by scores.

- **[function]** **measure-confusion** *truths predictions &key (test #'eql) truth-key prediction-key weight*

  Create a `confusion-matrix` from `truths` and `predictions`. `truths` (keyed by `truth-key`) is a sequence of class labels compared with `test` to another sequence of class labels in `predictions` (keyed by `prediction-key`). If `weight` is non-nil, then it is a function that returns the weight of an element of `truths`. Weighted cases add their weight to both counts (returned as the first and second values).

  Note how the returned confusion matrix can be added to another with `add-to-counter`.

7.3 Classification Counters

- **[class]** **classification-accuracy-counter** *basic-counter*

  A `basic-counter` with "acc." as its `:type` attribute and a `print-object` method that prints percentages.

- **[class]** **cross-entropy-counter** *basic-counter*

  A `basic-counter` with "xent" as its `:type` attribute.

7.3.1 Confusion Matrices

- **[class]** **confusion-matrix**

  A confusion matrix keeps count of classification results. The correct class is called `target' and the output of the classifier is called`prediction'.

- **[function]** **make-confusion-matrix** *&key (test #'eql)*

  Classes are compared with `test`.

- **[generic-function]** **sort-confusion-classes** *matrix classes*

  Return a list of `classes` sorted for presentation purposes.

- **[generic-function]** **confusion-class-name** *matrix class*

  Name of `class` for presentation purposes.

- **[generic-function]** **confusion-count** *matrix target prediction*

- **[generic-function]** **map-confusion-matrix** *fn matrix*

  Call `fn` with `target`, `prediction`, `count` paramaters for each cell in the confusion matrix. Cells with a zero count may be ommitted.

- **[generic-function]** **confusion-matrix-classes** *matrix*

  A list of all classes. The default is to collect classes from the counts. This can be overridden if, for instance, some classes are not present in the results.

- **[function]** **confusion-matrix-accuracy** *matrix &key filter*

  Return the overall accuracy of the results in `matrix`. It's computed as the number of correctly classified cases (hits) divided by the name of cases. Return the number of hits and the number of cases as the second and third value. If `filter` function is given, then call it with the target and the prediction of the cell. Disregard cell for which `filter` returns `nil`.

  Precision and recall can be easily computed by giving the right filter, although those are provided in separate convenience functions.

- **[function]** **confusion-matrix-precision** *matrix prediction*

  Return the accuracy over the cases when the classifier said `prediction`.

- **[function]** **confusion-matrix-recall** *matrix target*

  Return the accuracy over the cases when the correct class is `target`.

- **[function]** **add-confusion-matrix** *matrix result-matrix*

  Add `matrix` into `result-matrix`.

8 Features

[in package MGL-CORE]

8.1 Feature Selection

The following *scoring functions* all return an `equal` hash table that maps features to scores.

- **[function]** **count-features** *documents mapper &key (key #'identity)*

  Return scored features as an `equal` hash table whose keys are features of `documents` and values are counts of occurrences of features. `mapper` takes a function and a document and calls function with features of the document.

```
(sort (alexandria:hash-table-alist
       (count-features '(("hello" "world")
                         ("this" "is" "our" "world"))
                       (lambda (fn document)
                         (map nil fn document))))
      #'string< :key #'car)
=> (("hello" . 1) ("is" . 1) ("our" . 1) ("this" . 1) ("world" . 2))
```

- **[function]** **feature-llrs** *documents mapper class-fn &key (classes (all-document-classes documents class-fn))*

  Return scored features as an `equal` hash table whose keys are features of `documents` and values are their log likelihood ratios. `mapper` takes a function and a document and calls function with features of the document.

```
(sort (alexandria:hash-table-alist
       (feature-llrs '((:a "hello" "world")
                       (:b "this" "is" "our" "world"))
                     (lambda (fn document)
                       (map nil fn (rest document)))
                     #'first))
      #'string< :key #'car)
=> (("hello" . 2.6032386) ("is" . 2.6032386) ("our" . 2.6032386)
    ("this" . 2.6032386) ("world" . 4.8428774e-8))
```

- **[function]** **feature-disambiguities** *documents mapper class-fn &key (classes (all-document-classes documents class-fn))*

  Return scored features as an `equal` hash table whose keys are features of `documents` and values are their *disambiguities*. `mapper` takes a function and a document and calls function with features of the document.

  From the paper 'Using Ambiguity Measure Feature Selection Algorithm for Support Vector Machine Classifier'.

8.2 Feature Encoding

Features can rarely be fed directly to algorithms as is, they need to be transformed in some way. Suppose we have a simple language model that takes a single word as input and predicts the next word. However, both input and output is to be encoded as float vectors of length 1000. What we do is find the top 1000 words by some measure (see Feature Selection) and associate these words with the integers in [0..999] (this is `encode`ing). By using for example one-hot encoding, we translate a word into a float vector when passing in the input. When the model outputs the probability distribution of the next word, we find the index of the max and find the word associated with it (this is `decode`ing)

- **[generic-function]** **encode** *encoder decoded*

  Encode `decoded` with `encoder`. This interface is generic enough to be almost meaningless. See `encoder/decoder` for a simple, `mgl-nlp:bag-of-words-encoder` for a slightly more involved example.

  If `encoder` is a function designator, then it's simply `funcall`ed with `decoded`.

- **[generic-function]** **decode** *decoder encoded*

  Decode `encoded` with `encoder`. For an `decoder` / `encoder` pair, `(decode decoder (encode encoder object))` must be equal in some sense to `object`.

  If `decoder` is a function designator, then it's simply `funcall`ed with `encoded`.

- **[class]** **encoder/decoder**

  Implements O(1) `encode` and `decode` by having an internal decoded-to-encoded and an encoded-to-decoded `equal` hash table. `encoder/decoder` objects can be saved and loaded (see Persistence) as long as the elements in the hash tables have read/write consitency.

```
(let ((indexer
        (make-indexer
         (alexandria:alist-hash-table '(("I" . 3) ("me" . 2) ("mine" . 1)))
         2)))
  (values (encode indexer "I")
          (encode indexer "me")
          (encode indexer "mine")
          (decode indexer 0)
          (decode indexer 1)
          (decode indexer 2)))
=> 0
=> 1
=> NIL
=> "I"
=> "me"
=> NIL
```

- **[function]** **make-indexer** *scored-features n &key (start 0) (class 'encoder/decoder)*

  Take the top `n` features from `scored-features` (see Feature Selection), assign indices to them starting from `start`. Return an `encoder/decoder` (or another `class`) that converts between objects and indices.

Also see Bag of Words.

9 Gradient Based Optimization

[in package MGL-OPT]

We have a real valued, differentiable function F and the task is to find the parameters that minimize its value. Optimization starts from a single point in the parameter space of F, and this

single point is updated iteratively based on the gradient and value of F at or around the current point.

Note that while the stated problem is that of global optimization, for non-convex functions, most algorithms will tend to converge to a local optimum.

Currently, there are two optimization algorithms: Gradient Descent (with several variants) and Conjugate Gradient both of which are first order methods (they do not need second order gradients) but more can be added with the Extension API.

- **[function]** **minimize** *optimizer gradient-source &key (weights (list-segments gradient-source)) (dataset *infinitely-empty-dataset*)*

  Minimize the value of the real valued function represented by `gradient-source` by updating some of its parameters in `weights` (a `mat` or a sequence of `mat`s). Return `weights`. `dataset` (see Datasets) is a set of unoptimized parameters of the same function. For example, `weights` may be the weights of a neural network while `dataset` is the training set consisting of inputs suitable for `set-input`. The default `dataset`, (`*infinitely-empty-dataset*`) is suitable for when all parameters are optimized, so there is nothing left to come from the environment.

  Optimization terminates if `dataset` is a sampler and it runs out or when some other condition met (see `termination`, for example). If `dataset` is a `sequence`, then it is reused over and over again.

  Examples for various optimizers are provided in Gradient Descent and Conjugate Gradient.

9.1 Iterative Optimizer

- **[class]** **iterative-optimizer**

  An abstract base class of Gradient Descent and Conjugate Gradient based optimizers that iterate over instances until a termination condition is met.

- **[reader]** **n-instances** *iterative-optimizer (:n-instances = 0)*

  The number of instances this optimizer has seen so far. Incremented automatically during optimization.

- **[accessor]** **termination** *iterative-optimizer (:termination = nil)*

  If a number, it's the number of instances to train on in the sense of `n-instances`. If `n-instances` is equal or greater than this value optimization stops. If `termination` is `nil`, then optimization will continue. If it is `t`, then optimization will stop. If it is a function of no arguments, then its return value is processed as if it was returned by `termination`.

- **[accessor]** **on-optimization-started** *iterative-optimizer (:on-optimization-started = nil)*

  An event hook with parameters `(optimizer gradient-source n-instances)`. Called after initializations are performed (INITIALIZE-OPTIMIZER, *INITIALIZE-GRADIENT-SOURCE*) but before optimization is started.

- **[accessor]** **on-optimization-finished** *iterative-optimizer (:on-optimization-finished = nil)*

An event hook with parameters `(optimizer gradient-source n-instances)`. Called when optimization has finished.

- **[accessor]** **on-n-instances-changed** *iterative-optimizer (:on-n-instances-changed = nil)*

  An event hook with parameters `(optimizer gradient-source n-instances)`. Called when optimization of a batch of instances is done and `n-instances` is incremented.

Now let's discuss a few handy utilities.

- **[function]** **monitor-optimization-periodically** *optimizer periodic-fns*

  For each periodic function in the list of `periodic-fns`, add a monitor to `optimizer`'s `on-optimization-started`, `on-optimization-finished` and `on-n-instances-changed` hooks. The monitors are simple functions that just call each periodic function with the event parameters (`optimizer gradient-source n-instances`). Return `optimizer`.

  To log and reset the monitors of the gradient source after every 1000 instances seen by `optimizer`:

  ```
  (monitor-optimization-periodically optimizer
                                     '((:fn log-my-test-error
                                        :period 2000)
                                       (:fn reset-optimization-monitors
                                        :period 1000
                                        :last-eval 0)))
  ```

  Note how we don't pass it's allowed to just pass the initargs for a `periodic-fn` instead of `periodic-fn` itself. The `:last-eval` 0 bit prevents `reset-optimization-monitors` from being called at the start of the optimization when the monitors are empty anyway.

- **[generic-function]** **reset-optimization-monitors** *optimizer gradient-source*

  Report the state of `monitors` of `optimizer` and `gradient-source` and reset their counters. See `monitor-optimization-periodically` for an example of how this is used.

- **[method]** **reset-optimization-monitors** *(optimizer iterative-optimizer) gradient-source*

  Log the counters of the monitors of `optimizer` and `gradient-source` and reset them.

- **[generic-function]** **report-optimization-parameters** *optimizer gradient-source*

  A utility that's often called at the start of optimization (from `on-optimization-started`). The default implementation logs the description of `gradient-source` (as in `describe`) and `optimizer` and calls `log-mat-room`.

9.2 Cost Function

The function being minimized is often called the *cost* or the *loss* function.

- **[generic-function]** **cost** *model*

  Return the value of the cost function being minimized. Calling this only makes sense in the context of an ongoing optimization (see `minimize`). The cost is that of a batch of instances.

- **[function]** **make-cost-monitors** *model &key operation-mode attributes*

  Return a list of `monitor` objects, each associated with one `basic-counter` with attribute `:type` "cost". Implemented in terms of `make-cost-monitors*`.

- **[generic-function]** **make-cost-monitors*** *model operation-mode attributes*

  Identical to `make-cost-monitors` bar the keywords arguments. Specialize this to add to support for new model types.

9.3 Gradient Descent

[in package MGL-GD]

Gradient descent is a first-order optimization algorithm. Relying completely on first derivatives, it does not even evaluate the function to be minimized. Let's see how to minimize a numerical lisp function with respect to some of its parameters.

```
(cl:defpackage :mgl-example-sgd
  (:use #:common-lisp #:mgl))

(in-package :mgl-example-sgd)

;;; Create an object representing the sine function.
(defparameter *diff-fn-1*
  (make-instance 'mgl-diffun:diffun
                 :fn #'sin
                 ;; We are going to optimize its only parameter.
                 :weight-indices '(0)))

;;; Minimize SIN. Note that there is no dataset involved because all
;;; parameters are being optimized.
(minimize (make-instance 'sgd-optimizer :termination 1000)
          *diff-fn-1*
          :weights (make-mat 1))
;;; => A MAT with a single value of about -pi/2.

;;; Create a differentiable function for f(x,y)=(x-y)^2. X is a
;;; parameter whose values come from the DATASET argument passed to
;;; MINIMIZE. Y is a parameter to be optimized (a 'weight').
(defparameter *diff-fn-2*
  (make-instance 'mgl-diffun:diffun
                 :fn (lambda (x y)
                       (expt (- x y) 2))
                 :parameter-indices '(0)
                 :weight-indices '(1)))

;;; Find the Y that minimizes the distance from the instances
;;; generated by the sampler.
(minimize (make-instance 'sgd-optimizer :batch-size 10)
          *diff-fn-2*
          :weights (make-mat 1)
          :dataset (make-instance 'function-sampler
```

```
                                  :generator (lambda ()
                                               (list (+ 10
                                                        (gaussian-random-1))))
                                  :max-n-samples 1000))
;;; => A MAT with a single value of about 10, the expected value of
;;; the instances in the dataset.

;;; The dataset can be a SEQUENCE in which case we'd better set
;;; TERMINATION else optimization would never finish.
(minimize (make-instance 'sgd-optimizer :termination 1000)
          *diff-fn-2*
          :weights (make-mat 1)
          :dataset '((0) (1) (2) (3) (4) (5)))
;;; => A MAT with a single value of about 2.5.
```

We are going to see a number of accessors for optimizer paramaters. In general, it's allowed to `setf` real slot accessors (as opposed to readers and writers) at any time during optimization and so is defining a method on an optimizer subclass that computes the value in any way. For example, to decay the learning rate on a per mini-batch basis:

```
(defmethod learning-rate ((optimizer my-sgd-optimizer))
  (* (slot-value optimizer 'learning-rate)
     (expt 0.998
           (/ (n-instances optimizer) 60000))))
```

9.3.1 Batch Based Optimizers

First let's see everything common to all batch based optimizers, then discuss SGD Optimizer, Adam Optimizer and Normalized Batch Optimizer. All batch based optimizers are `iterative-optimizer`s, so see Iterative Optimizer too.

- **[class]** **batch-gd-optimizer** *iterative-optimizer*

  Another abstract base class for gradient based optimizers tath updates all weights simultaneously after chewing through `batch-size` inputs. See subclasses `sgd-optimizer`, `adam-optimizer` and `normalized-batch-gd-optimizer`.

  `per-weight-batch-gd-optimizer` may be a better choice when some weights can go unused for instance due to missing input values.

- **[accessor]** **batch-size** *gd-optimizer (:batch-size = 1)*

  After having gone through `batch-size` number of inputs, weights are updated. With `batch-size` 1, one gets Stochastics Gradient Descent. With `batch-size` equal to the number of instances in the dataset, one gets standard, 'batch' gradient descent. With `batch-size` between these two extremes, one gets the most practical 'mini-batch' compromise.

- **[accessor]** **learning-rate** *gd-optimizer (:learning-rate = 0.1)*

  This is the step size along the gradient. Decrease it if optimization diverges, increase it if it doesn't make progress.

- **[accessor]** **momentum** *gd-optimizer (:momentum = 0)*

  A value in the [0, 1) interval. `momentum` times the previous weight change is added to the gradient. 0 means no momentum.

- **[reader]** **momentum-type** *gd-optimizer (:momentum-type = :normal)*

  One of `:normal`, `:nesterov` or `:none`. For pure optimization Nesterov's momentum may be better, but it may also increases chances of overfitting. Using `:none` is equivalent to 0 momentum, but it also uses less memory. Note that with `:none`, `momentum` is ignored even it it is non-zero.

- **[accessor]** **weight-decay** *gd-optimizer (:weight-decay = 0)*

  An L2 penalty. It discourages large weights, much like a zero mean gaussian prior. `weight-decay` * WEIGHT is added to the gradient to penalize large weights. It's as if the function whose minimum is sought had `weight-decay`*sum_i{0.5 * WEIGHT_i^2} added to it.

- **[accessor]** **weight-penalty** *gd-optimizer (:weight-penalty = 0)*

  An L1 penalty. It encourages sparsity. `sign`(WEIGHT) * `weight-penalty` is added to the gradient pushing the weight towards negative infinity. It's as if the function whose minima is sought had `weight-penalty`*sum_i{abs(WEIGHT_i)} added to it. Putting it on feature biases consitutes a sparsity constraint on the features.

- **[reader]** **use-segment-derivatives-p** *gd-optimizer (:use-segment-derivatives-p = nil)*

  Save memory if both the gradient source (the model being optimized) and the optimizer support this feature. It works like this: the accumulator into which the gradient source is asked to place the derivatives of a segment will be `segment-derivatives` of the segment. This allows the optimizer not to allocate an accumulator matrix into which the derivatives are summed.

- **[accessor]** **after-update-hook** *gd-optimizer (:after-update-hook = nil)*

  A list of functions with no arguments called after each weight update.

- **[accessor]** **before-update-hook** *batch-gd-optimizer (:before-update-hook = nil)*

  A list of functions of no parameters. Each function is called just before a weight update takes place (after accumulated gradients have been divided the length of the batch). Convenient to hang some additional gradient accumulating code on.

SGD Optimizer

- **[class]** **sgd-optimizer** *batch-gd-optimizer*

  With `batch-size` 1 this is Stochastic Gradient Descent. With higher batch sizes, one gets mini-batch and Batch Gradient Descent.

  Assuming that `accumulator` has the sum of gradients for a mini-batch, the weight update looks like this:

$$\Delta_w^{t+1} = momentum * \Delta_w^t + \frac{accumulator}{batchsize} + l_2 w + l_1 sign(w)$$

$$w^{t+1} = w^t - learningrate * \Delta_w,$$

which is the same as the more traditional formulation:

$$\Delta_w^{t+1} = momentum * \Delta_w^t + learningrate * \left(\frac{\frac{df}{dw}}{batchsize} + l_2 w + l_1 sign(w) \right)$$

$$w^{t+1} = w^t - \Delta_w,$$

but the former works better when batch size, momentum or learning rate change during the course of optimization. The above is with normal momentum, Nesterov's momentum (see `momentum-type`) momentum is also available.

See Batch Based Optimizers for the description of the various options common to all batch based optimizers.

Adam Optimizer

- **[class]** **adam-optimizer** *batch-gd-optimizer*

  Adam is a first-order stochasistic gradient descent optimizer. It maintains an internal estimation for the mean and raw variance of each derivative as exponential moving averages. The step it takes is basically `M/(sqrt(V)+E)` where `m` is the estimated mean, `v` is the estimated variance, and `e` is a small adjustment factor to prevent the gradient from blowing up. See version 5 of the paper for more.

  Note that using momentum is not supported with Adam. In fact, an error is signalled if it's not `:none`.

  See Batch Based Optimizers for the description of the various options common to all batch based optimizers.

- **[accessor]** **learning-rate** *adam-optimizer (= 2.0e-4)*

  Same thing as `learning-rate` but with the default suggested by the Adam paper.

- **[accessor]** **mean-decay** *adam-optimizer (:mean-decay = 0.9)*

  A number between 0 and 1 that determines how fast the estimated mean of derivatives is updated. 0 basically gives you RMSPROP (if `variance-decay` is not too large) or AdaGrad (if `variance-decay` is close to 1 and the learning rate is annealed. This is β_1 in the paper.

- **[accessor]** **mean-decay-decay** *adam-optimizer (:mean-decay-decay = (- 1 1.0d-7))*

  A value that should be close to 1. `mean-decay` is multiplied by this value after each update. This is λ in the paper.

- **[accessor]** **variance-decay** *adam-optimizer (:variance-decay = 0.999)*

  A number between 0 and 1 that determines how fast the estimated variance of derivatives is updated. This is β_2 in the paper.

- **[accessor]** **variance-adjustment** *adam-optimizer (:variance-adjustment = 1.0d-7)*

  Within the bowels of adam, the estimated mean is divided by the square root of the estimated variance (per weight) which can lead to numerical problems if the denominator is near zero. To avoid this, `variance-adjustment`, which should be a small positive number, is added to the denominator. This is `epsilon` in the paper.

Normalized Batch Optimizer

- **[class]** **normalized-batch-gd-optimizer** *batch-gd-optimizer*

  Like `batch-gd-optimizer` but keeps count of how many times each weight was used in the batch and divides the accumulated gradient by this count instead of dividing by `n-instances-in-batch`. This only makes a difference if there are missing values in the learner that's being trained. The main feature that distuinguishes this class from `per-weight-batch-gd-optimizer` is that batches end at same time for all weights.

- **[accessor]** **n-weight-uses-in-batch** *normalized-batch-gd-optimizer*

  Number of uses of the weight in its current batch.

9.3.2 Segmented GD Optimizer

- **[class]** **segmented-gd-optimizer** *iterative-optimizer*

  An optimizer that delegates training of segments to other optimizers. Useful to delegate training of different segments to different optimizers (capable of working with segmentables) or simply to not train all segments.

- **[reader]** **segmenter** *segmented-gd-optimizer (:segmenter)*

  When this optimizer is initialized it loops over the segment of the learner with `map-segments`. `segmenter` is a function that is called with each segment and returns an optimizer or `nil`. Several segments may be mapped to the same optimizer. After the segment->optimizer mappings are collected, each optimizer is initialized by INITIALIZE-OPTIMIZER with the list of segments mapped to it.

- **[reader]** **segments** *segmented-gd-optimizer*

`segmented-gd-optimizer` inherits from `iterative-optimizer`, so see Iterative Optimizer too.

9.3.3 Per-weight Optimization

- **[class]** **per-weight-batch-gd-optimizer** *iterative-optimizer*

  This is much like Batch Based Optimizers but it is more clever about when to update weights. Basically every weight has its own batch independent from the batches of others. This

has desirable properties. One can for example put two neural networks together without adding any connections between them and the learning will produce results equivalent to the separated case. Also, adding inputs with only missing values does not change anything.

Due to its very non-batch nature, there is no CUDA implementation of this optimizer.

- **[accessor]** **n-weight-uses-in-batch** *per-weight-batch-gd-optimizer*

  Number of uses of the weight in its current batch.

9.3.4 Utilities

- **[function]** **clip-l2-norm** *mats l2-upper-bound &key callback*

  Scale `mats` so that their L_2 norm does not exceed `l2-upper-bound`.

  Compute the norm of of `mats` as if they were a single vector. If the norm is greater than `l2-upper-bound`, then scale each matrix destructively by the norm divided by `l2-upper-bound` and if non-`nil` call the function `callback` with the scaling factor.

- **[function]** **arrange-for-clipping-gradients** *batch-gd-optimizer l2-upper-bound &key callback*

  Make it so that the norm of the batch normalized gradients accumulated by `batch-gd-optimizer` is clipped to `l2-upper-bound` before every update. See `clip-l2-norm`.

9.4 Conjugate Gradient

[in package MGL-CG]

Conjugate gradient is a first-order optimization algorithm. It's more advanced than gradient descent as it does line searches which unfortunately also makes it unsuitable for non-deterministic functions. Let's see how to minimize a numerical lisp function with respect to some of its parameters.

```
;;; Create an object representing the sine function.
(defparameter *diff-fn-1*
  (make-instance 'mgl-diffun:diffun
                 :fn #'sin
                 ;; We are going to optimize its only parameter.
                 :weight-indices '(0)))

;;; Minimize SIN. Note that there is no dataset involved because all
;;; parameters are being optimized.
(minimize (make-instance 'cg-optimizer
                         :batch-size 1
                         :termination 1)
          *diff-fn-1*
          :weights (make-mat 1))
;;; => A MAT with a single value of about -pi/2.

;;; Create a differentiable function for f(x,y)=(x-y)^2. X is a
;;; parameter whose values come from the DATASET argument passed to
```

```
;;; MINIMIZE. Y is a parameter to be optimized (a 'weight').
(defparameter *diff-fn-2*
  (make-instance 'mgl-diffun:diffun
                 :fn (lambda (x y)
                       (expt (- x y) 2))
                 :parameter-indices '(0)
                 :weight-indices '(1)))

;;; Find the Y that minimizes the distance from the instances
;;; generated by the sampler.
(minimize (make-instance 'cg-optimizer :batch-size 10)
          *diff-fn-2*
          :weights (make-mat 1)
          :dataset (make-instance 'function-sampler
                                  :generator (lambda ()
                                               (list (+ 10
                                                        (gaussian-random-1))))
                                  :max-n-samples 1000))
;;; => A MAT with a single value of about 10, the expected value of
;;; the instances in the dataset.

;;; The dataset can be a SEQUENCE in which case we'd better set
;;; TERMINATION else optimization would never finish. Note how a
;;; single epoch suffices.
(minimize (make-instance 'cg-optimizer :termination 6)
          *diff-fn-2*
          :weights (make-mat 1)
          :dataset '((0) (1) (2) (3) (4) (5)))
;;; => A MAT with a single value of about 2.5.
```

- **[function]** **cg** *fn w &key (max-n-line-searches *default-max-n-line-searches*) (max-n-evaluations-per-line-search *default-max-n-evaluations-per-line-search*) (max-n-evaluations *default-max-n-evaluations*) (sig *default-sig*) (rho *default-rho*) (int *default-int*) (ext *default-ext*) (ratio *default-ratio*) spare-vectors*

  `cg-optimizer` passes each batch of data to this function with its `cg-args` passed on.

  Minimize a differentiable multivariate function with conjugate gradient. The Polak-Ribiere flavour of conjugate gradients is used to compute search directions, and a line search using quadratic and cubic polynomial approximations and the Wolfe-Powell stopping criteria is used together with the slope ratio method for guessing initial step sizes. Additionally a bunch of checks are made to make sure that exploration is taking place and that extrapolation will not be unboundedly large.

  `fn` is a function of two parameters: `weights` and `derivatives`. `weights` is a `mat` of the same size as `w` that is where the search start from. `derivatives` is also a `mat` of that size and it is where `fn` shall place the partial derivatives. `fn` returns the value of the function that is being minimized.

  `cg` performs a number of line searches and invokes `fn` at each step. A line search invokes `fn` at most `max-n-evaluations-per-line-search` number of times and can succeed in

improving the minimum by the sufficient margin or it can fail. Note, the even a failed line search may improve further and hence change the weights it's just that the improvement was deemed too small. `cg` stops when either:

- two line searches fail in a row
- `max-n-line-searches` is reached
- `max-n-evaluations` is reached

`cg` returns a `mat` that contains the best weights, the minimum, the number of line searches performed, the number of succesful line searches and the number of evaluations.

When using `max-n-evaluations` remember that there is an extra evaluation of `fn` before the first line search.

`spare-vectors` is a list of preallocated `mat`s of the same size as `w`. Passing 6 of them covers the current need of the algorithm and it will not cons up vectors of size `w` at all.

`note`: If the function terminates within a few iterations, it could be an indication that the function values and derivatives are not consistent (ie, there may be a bug in the implementation of `fn` function).

`sig` and `rho` are the constants controlling the Wolfe-Powell conditions. `sig` is the maximum allowed absolute ratio between previous and new slopes (derivatives in the search direction), thus setting `sig` to low (positive) values forces higher precision in the line-searches. `rho` is the minimum allowed fraction of the expected (from the slope at the initial point in the linesearch). Constants must satisfy 0 < `rho` < `sig` < 1. Tuning of `sig` (depending on the nature of the function to be optimized) may speed up the minimization; it is probably not worth playing much with `rho`.

- **[variable]** ***default-int*** *0.1*

  Don't reevaluate within `int` of the limit of the current bracket.

- **[variable]** ***default-ext*** *3*

  Extrapolate maximum `ext` times the current step-size.

- **[variable]** ***default-sig*** *0.1*

  `sig` and `rho` are the constants controlling the Wolfe-Powell conditions. `sig` is the maximum allowed absolute ratio between previous and new slopes (derivatives in the search direction), thus setting `sig` to low (positive) values forces higher precision in the line-searches.

- **[variable]** ***default-rho*** *0.05*

  `rho` is the minimum allowed fraction of the expected (from the slope at the initial point in the linesearch). Constants must satisfy 0 < `rho` < `sig` < 1.

- **[variable]** ***default-ratio*** *10*

  Maximum allowed slope ratio.

- **[variable]** ***default-max-n-line-searches*** *nil*

- **[variable]** ***default-max-n-evaluations-per-line-search*** *20*
- **[variable]** ***default-max-n-evaluations*** *nil*
- **[class]** **cg-optimizer** *iterative-optimizer*

  Updates all weights simultaneously after chewing through `batch-size` inputs.
- **[accessor]** **batch-size** *cg-optimizer (:batch-size)*

  After having gone through `batch-size` number of instances, weights are updated. Normally, `cg` operates on all available data, but it may be useful to introduce some noise into the optimization to reduce overfitting by using smaller batch sizes. If `batch-size` is not set, it is initialized to the size of the dataset at the start of optimization.
- **[accessor]** **cg-args** *cg-optimizer (:cg-args = 'nil)*
- **[accessor]** **on-cg-batch-done** *cg-optimizer (:on-cg-batch-done = nil)*

  An event hook called when processing a conjugate gradient batch is done. The handlers on the hook are called with 8 arguments:

  ```
  (optimizer gradient-source instances
   best-w best-f n-line-searches
   n-succesful-line-searches n-evaluations)
  ```

  The latter 5 of which are the return values of the `cg` function.
- **[generic-function]** **log-cg-batch-done** *optimizer gradient-source instances best-w best-f n-line-searches n-succesful-line-searches n-evaluations*

  This is a function can be added to `on-cg-batch-done`. The default implementation simply logs the event arguments.
- **[reader]** **segment-filter** *cg-optimizer (:segment-filter = (constantly t))*

  A predicate function on segments that filters out uninteresting segments. Called from `initialize-optimizer*`.

9.5 Extension API

9.5.1 Implementing Optimizers

The following generic functions must be specialized for new optimizer types.

- **[generic-function]** **minimize*** *optimizer gradient-source weights dataset*

  Called by `minimize` after `initialize-optimizer*` and `initialize-gradient-source*`, this generic function is the main extension point for writing optimizers.
- **[generic-function]** **initialize-optimizer*** *optimizer gradient-source weights dataset*

Called automatically before training starts, this function sets up `optimizer` to be suitable for optimizing `gradient-source`. It typically creates appropriately sized accumulators for the gradients.

- **[generic-function]** **segments** *optimizer*

  Several weight matrices known as *segments* can be optimized by a single optimizer. This function returns them as a list.

The rest are just useful for utilities for implementing optimizers.

- **[function]** **terminate-optimization-p** *n-instances termination*

  Utility function for subclasses of `iterative-optimizer`. It returns whether optimization is to be terminated based on `n-instances` and `termination` that are values of the respective accessors of `iterative-optimizer`.

- **[function]** **set-n-instances** *optimizer gradient-source n-instances*

  Set `n-instances` of `optimizer` and fire `on-n-instances-changed`. `iterative-optimizer` subclasses must call this to increment `n-instances`.

- **[class]** **segment-set**

  This is a utility class for optimizers that have a list of `segments` and (the weights being optimized) is able to copy back and forth between those segments and a single `mat` (the accumulator).

- **[reader]** **segments** *segment-set (:segments)*

  A list of weight matrices.

- **[reader]** **size** *segment-set*

  The sum of the sizes of the weight matrices of `segments`.

- **[macro]** **do-segment-set** *(segment &optional start) segment-set &body body*

  Iterate over `segments` in `segment-set`. If `start` is specified, the it is bound to the start index of `segment` within `segment-set`. The start index is the sum of the sizes of previous segments.

- **[function]** **segment-set<-mat** *segment-set mat*

  Copy the values of `mat` to the weight matrices of `segment-set` as if they were concatenated into a single `mat`.

- **[function]** **segment-set->mat** *segment-set mat*

  Copy the values of `segment-set` to `mat` as if they were concatenated into a single `mat`.

9.5.2 Implementing Gradient Sources

Weights can be stored in a multitude of ways. Optimizers need to update weights, so it is assumed that weights are stored in any number of `mat` objects called segments.

The generic functions in this section must all be specialized for new gradient sources except where noted.

- **[generic-function]** **map-segments** *fn gradient-source*

  Apply `fn` to each segment of `gradient-source`.

- **[generic-function]** **map-segment-runs** *fn segment*

  Call `fn` with start and end of intervals of consecutive indices that are not missing in `segment`. Called by optimizers that support partial updates. The default implementation assumes that all weights are present. This only needs to be specialized if one plans to use an optimizer that knows how to deal unused/missing weights such as `mgl-gd:normalized-batch-gd-optimizer` and `optimizer` `mgl-gd:per-weight-batch-gd-optimizer`.

- **[generic-function]** **segment-weights** *segment*

  Return the weight matrix of `segment`. A segment doesn't need to be a `mat` object itself. For example, it may be a `mgl-bm:chunk` of a `mgl-bm:bm` or a `mgl-bp:lump` of a `mgl-bp:bpn` whose `nodes` slot holds the weights.

- **[method]** **segment-weights** *(mat mat)*

  When the segment is really a `mat`, then just return it.

- **[generic-function]** **segment-derivatives** *segment*

  Return the derivatives matrix of `segment`. A segment doesn't need to be a `mat` object itself. For example, it may be a `mgl-bm:chunk` of a `mgl-bm:bm` or a `mgl-bp:lump` of a `mgl-bp:bpn` whose DERIVATIVES slot holds the gradient.

- **[function]** **list-segments** *gradient-source*

  A utility function that returns the list of segments from `map-segments` on `gradient-source`.

- **[generic-function]** **initialize-gradient-source*** *optimizer gradient-source weights dataset*

  Called automatically before `minimize*` is called, this function may be specialized if `gradient-source` needs some kind of setup.

- **[method]** **initialize-gradient-source*** *optimizer gradient-source weights dataset*

  The default method does nothing.

- **[generic-function]** **accumulate-gradients*** *gradient-source sink batch multiplier valuep*

  Add `multiplier` times the sum of first-order gradients to accumulators of `sink` (normally accessed with `do-gradient-sink`) and if `valuep`, return the sum of values of the function being optimized for a `batch` of instances. `gradient-source` is the object representing the function being optimized, `sink` is gradient sink.

  Note the number of instances in `batch` may be larger than what `gradient-source` process in one go (in the sense of say, `max-n-stripes`), so `do-batches-for-model` or something like (`group batch max-n-stripes`) can be handy.

9.5.3 Implementing Gradient Sinks

Optimizers call `accumulate-gradients*` on gradient sources. One parameter of `accumulate-gradients*` is the `sink`. A gradient sink knows what accumulator matrix (if any) belongs to a segment. Sinks are defined entirely by `map-gradient-sink`.

- **[generic-function]** **map-gradient-sink** *fn sink*

  Call `fn` of lambda list (`segment accumulator`) on each segment and their corresponding accumulator `mat` in `sink`.

- **[macro]** **do-gradient-sink** *((segment accumulator) sink) &body body*

  A convenience macro on top of `map-gradient-sink`.

10 Differentiable Functions

[in package MGL-DIFFUN]

- **[class]** **diffun**

```
`diffun` dresses a lisp function (in its [`fn`][f491] slot) as
a gradient source (see [Implementing Gradient Sources][c58b]), which
allows it to be used in [`minimize`][46a4]. See the examples in
[Gradient Descent][10e7] and [Conjugate Gradient][83e6].
```

- **[reader]** **fn** *diffun (:fn)*

  A real valued lisp function. It may have any number of parameters.

- **[reader]** **parameter-indices** *diffun (:parameter-indices = nil)*

  The list of indices of parameters that we don't optimize. Values for these will come from the DATASET argument of `minimize`.

- **[reader]** **weight-indices** *diffun (:weight-indices = nil)*

  The list of indices of parameters to be optimized, the values of which will come from the `weights` argument of `minimize`.

11 Backpropagation Neural Networks

[in package MGL-BP]

11.1 Backprop Overview

Backpropagation Neural Networks are just functions with lots of parameters called *weights* and a layered structure when presented as a computational graph. The network is trained to `minimize` some kind of *loss function* whose value the network computes.

In this implementation, a `bpn` is assembled from several `lump`s (roughly corresponding to layers). Both feed-forward and recurrent neural nets are supported (`fnn` and `rnn`, respectively). `bpn`s

can contain not only lumps but other bpns, too. As we see, networks are composite objects and the abstract base class for composite and simple parts is called clump.

- **[class] clump**

  A clump is a lump or a bpn. It represents a differentiable function. Arguments of clumps are given during instantiation. Some arguments are clumps themselves so they get permenantly wired together like this:

```
(->v*m (->input :size 10 :name 'input)
       (->weight :dimensions '(10 20) :name 'weight)
       :name 'activation)
```

  The above creates three clumps: the vector-matrix multiplication clumps called activation which has a reference to its operands: input and weight. Note that the example just defines a function, no actual computation has taken place, yet.

  This wiring of clumps is how one builds feed-forward nets (fnn) or recurrent neural networks (rnn) that are clumps themselves so one can build nets in a hiearchical style if desired. Non-composite clumps are called lump (note the loss of c that stands for composite). The various lump subtypes correspond to different layer types (->sigmoid, ->dropout, ->relu, ->tanh, etc).

At this point, you may want to jump ahead to get a feel for how things work by reading the fnn Tutorial.

11.2 Clump API

These are mostly for extension purposes. About the only thing needed from here for normal operation is nodes when clamping inputs or extracting predictions.

- **[generic-function] stripedp** *clump*

  For efficiency, forward and backprop phases do their stuff in batch mode: passing a number of instances through the network in batches. Thus clumps must be able to store values of and gradients for each of these instances. However, some clumps produce the same result for each instance in a batch. These clumps are the weights, the parameters of the network. stripedp returns true iff clump does not represent weights (i.e. it's not a ->weight).

  For striped clumps, their nodes and derivatives are mat objects with a leading dimension (number of rows in the 2d case) equal to the number of instances in the batch. Non-striped clumps have no restriction on their shape apart from what their usage dictates.

- **[generic-function] nodes** *object*

  Returns a mgl-mat:mat object representing the state or result of object. The first dimension of the returned matrix is equal to the number of stripes.

clumps' nodes holds the result computed by the most recent forward. For ->input lumps, this is where input values shall be placed (see set-input). Currently, the matrix is always two dimensional but this restriction may go away in the future.

- **[generic-function]** **derivatives** *clump*

  Return the `mat` object representing the partial derivatives of the function `clump` computes. The returned partial derivatives were accumulated by previous `backward` calls.

  This matrix is shaped like the matrix returned by `nodes`.

- **[generic-function]** **forward** *clump*

  Compute the values of the function represented by `clump` for all stripes and place the results into `nodes` of `clump`.

- **[generic-function]** **backward** *clump*

  Compute the partial derivatives of the function represented by `clump` and add them to `derivatives` of the corresponding argument clumps. The `derivatives` of `clump` contains the sum of partial derivatives of all clumps by the corresponding output. This function is intended to be called after a `forward` pass.

  Take the `->sigmoid` clump for example when the network is being applied to a batch of two instances `x1` and `x2`. `x1` and `x2` are set in the `->input` lump X. The sigmoid computes `1/(1+exp(-x))` where `x` is its only argument clump.

  ```
  f(x) = 1/(1+exp(-x))
  ```

  When `backward` is called on the sigmoid lump, its `derivatives` is a 2x1 `mat` object that contains the partial derivatives of the loss function:

  ```
  dL(x1)/df
  dL(x2)/df
  ```

  Now the `backward` method of the sigmoid needs to add `dL(x1)/dx1` and `dL(x2)/dx2` to `derivatives` of `x`. Now, `dL(x1)/dx1 = dL(x1)/df * df(x1)/dx1` and the first term is what we have in `derivatives` of the sigmoid so it only needs to calculate the second term.

In addition to the above, clumps also have to support `size`, `n-stripes`, `max-n-stripes` (and the `setf` methods of the latter two) which can be accomplished just by inheriting from `bpn`, `fnn`, `rnn`, or a `lump`.

11.3 bpns

- **[class]** **bpn** *clump*

  Abstract base class for `fnn` and `rnn`.

- **[reader]** **n-stripes** *bpn (:n-stripes = 1)*

  The current number of instances the network has. This is automatically set to the number of instances passed to `set-input`, so it rarely has to be manipulated directly although it can be set. When set `n-stripes` of all `clumps` get set to the same value.

- **[reader]** **max-n-stripes** *bpn (:max-n-stripes = nil)*

The maximum number of instances the network can operate on in parallel. Within `build-fnn` or `build-rnn`, it defaults to `max-n-stripes` of that parent network, else it defaults to 1. When set `max-n-stripes` of all `clumps` get set to the same value.

- **[reader]** **clumps** *bpn (:clumps = (make-array 0 :element-type 'clump :adjustable t :fill-pointer t))*

  A topological sorted adjustable array with a fill pointer that holds the clumps that make up the network. Clumps are added to it by `add-clump` or, more often, automatically when within a `build-fnn` or `build-rnn`. Rarely needed, `find-clump` takes care of most uses.

- **[function]** **find-clump** *name bpn &key (errorp t)*

  Find the clump with `name` among `clumps` of `bpn`. As always, names are compared with `equal`. If not found, then return `nil` or signal and error depending on `errorp`.

- **[function]** **add-clump** *clump bpn*

  Add `clump` to `bpn`. `max-n-stripes` of `clump` gets set to that of `bpn`. It is an error to add a clump with a name already used by one of the `clumps` of `bpn`.

11.3.1 Training

`bpn`s are trained to minimize the loss function they compute. Before a `bpn` is passed to `minimize` (as its `gradient-source` argument), it must be wrapped in a `bp-learner` object. `bp-learner` has `monitors` slot which is used for example by `reset-optimization-monitors`.

Without the bells an whistles, the basic shape of training is this:

```
(minimize optimizer (make-instance 'bp-learner :bpn bpn)
          :dataset dataset)
```

- **[class]** **bp-learner**

- **[reader]** **bpn** *bp-learner (:bpn)*

  The `bpn` for which this `bp-learner` provides the gradients.

- **[accessor]** **monitors** *bp-learner (:monitors = nil)*

  A list of `monitor`s.

11.3.2 Monitoring

- **[function]** **monitor-bpn-results** *dataset bpn monitors*

  For every batch (of size `max-n-stripes` of `bpn`) of instances in `dataset`, set the batch as the next input with `set-input`, perform a `forward` pass and apply `monitors` to the `bpn` (with `apply-monitors`). Finally, return the counters of `monitors`. This is built on top of `monitor-model-results`.

- **[function]** **make-step-monitor-monitors** *rnn &key (counter-values-fn #'counter-raw-values) (make-counter #'make-step-monitor-monitor-counter)*

Return a list of monitors, one for every monitor in `step-monitors` of `rnn`. These monitors extract the results from their warp counterpairs with `counter-values-fn` and add them to their own counter that's created by `make-counter`. Wow. Ew. The idea is that one does something like this do monitor warped prediction:

```
(let ((*warp-time* t))
  (setf (step-monitors rnn)
        (make-cost-monitors rnn :attributes '(:event "warped pred.")))
  (monitor-bpn-results dataset rnn
                       ;; Just collect and reset the warp
                       ;; monitors after each batch of
                       ;; instances.
                       (make-step-monitor-monitors rnn)))
```

- **[generic-function]** **make-step-monitor-monitor-counter** *step-counter*

  In an `rnn`, `step-counter` aggregates results of all the time steps during the processing of instances in the current batch. Return a new counter into which results from `step-counter` can be accumulated when the processing of the batch is finished. The default implementation creates a copy of `step-counter`.

11.3.3 Feed-Forward Nets

`fnn` and `rnn` have a lot in common (see their common superclass, `bpn`). There is very limited functionality that's specific to `fnn`s so let's get them out of they way before we study a full example.

- **[class]** **fnn** *bpn*

  A feed-forward neural net (as opposed to a recurrent one, see `rnn`).

- **[macro]** **build-fnn** *(&key fnn (class 'fnn) initargs max-n-stripes name) &body clumps*

  Syntactic sugar to assemble `fnn`s from `clump`s. Like `let*`, it is a sequence of bindings (of symbols to `clump`s). The names of the clumps created default to the symbol of the binding. In case a clump is not bound to a symbol (because it was created in a nested expression), the local function `clump` can be used to find the clump with the given name in the fnn being built. Example:

```
(build-fnn ()
  (features (->input :size n-features))
  (biases (->weight :size n-features))
  (weights (->weight :size (* n-hiddens n-features)))
  (activations0 (->v*m :weights weights :x (clump 'features)))
  (activations (->+ :args (list biases activations0)))
  (output (->sigmoid :x activations)))
```

`fnn` Tutorial Hopefully this example from `example/digit-fnn.lisp` illustrates the concepts involved. If it's too dense despite the comments, then read up on Datasets, Gradient Based Optimization and come back.

```
(cl:defpackage :mgl-example-digit-fnn
  (:use #:common-lisp #:mgl))

(in-package :mgl-example-digit-fnn)

;;; There are 10 possible digits used as inputs ...
(defparameter *n-inputs* 10)
;;; and we want to learn the rule that maps the input digit D to (MOD
;;; (1+ D) 3).
(defparameter *n-outputs* 3)

;;; We define a feed-forward net to be able to specialize how inputs
;;; are translated by adding a SET-INPUT method later.
(defclass digit-fnn (fnn)
  ())

;;; Build a DIGIT-FNN with a single hidden layer of rectified linear
;;; units and a softmax output.
(defun make-digit-fnn (&key (n-hiddens 5))
  (build-fnn (:class 'digit-fnn)
    (input (->input :size *n-inputs*))
    (hidden-activation (->activation input :size n-hiddens))
    (hidden (->relu hidden-activation))
    (output-activation (->activation hidden :size *n-outputs*))
    (output (->softmax-xe-loss output-activation))))

;;; This method is called with batches of 'instances' (input digits in
;;; this case) by MINIMIZE and also by MONITOR-BPN-RESULTS before
;;; performing a forward pass (i.e. computing the value of the
;;; function represented by the network). Its job is to encode the
;;; inputs by populating rows of the NODES matrix of the INPUT clump.
;;;
;;; Each input is encoded as a row of zeros with a single 1 at index
;;; determined by the input digit. This is called one-hot encoding.
;;; The TARGET could be encoded the same way, but instead we use the
;;; sparse option supported by TARGET of ->SOFTMAX-XE-LOSS.
(defmethod set-input (digits (fnn digit-fnn))
  (let* ((input (nodes (find-clump 'input fnn)))
         (output-lump (find-clump 'output fnn)))
    (fill! 0 input)
    (loop for i upfrom 0
          for digit in digits
          do (setf (mref input i digit) 1))
    (setf (target output-lump)
          (mapcar (lambda (digit)
                    (mod (1+ digit) *n-outputs*))
                  digits))))

;;; Train the network by minimizing the loss (cross-entropy here) with
;;; stochastic gradient descent.
(defun train-digit-fnn ()
  (let ((optimizer
```

```
          ;; First create the optimizer for MINIMIZE.
          (make-instance 'segmented-gd-optimizer
                         :segmenter
                         ;; We train each weight lump with the same
                         ;; parameters and, in fact, the same
                         ;; optimizer. But it need not be so, in
                         ;; general.
                         (constantly
                          (make-instance 'sgd-optimizer
                                         :learning-rate 1
                                         :momentum 0.9
                                         :batch-size 100))))
        (fnn (make-digit-fnn)))
    ;; The number of instances the FNN can work with in parallel. It's
    ;; usually equal to the batch size or is a its divisor.
    (setf (max-n-stripes fnn) 50)
    ;; Initialize all weights randomly.
    (map-segments (lambda (weights)
                    (gaussian-random! (nodes weights) :stddev 0.01))
                  fnn)
    ;; Arrange for training and test error to be logged.
    (monitor-optimization-periodically
     optimizer '((:fn log-test-error :period 10000)
                 (:fn reset-optimization-monitors :period 1000)))
    ;; Finally, start the optimization.
    (minimize optimizer
              ;; Dress FNN in a BP-LEARNER and attach monitors for the
              ;; cost to it. These monitors are going to be logged and
              ;; reset after every 100 training instance by
              ;; RESET-OPTIMIZATION-MONITORS above.
              (make-instance 'bp-learner
                             :bpn fnn
                             :monitors (make-cost-monitors
                                        fnn :attributes `(:event "train")))
              ;; Training stops when the sampler runs out (after 10000
              ;; instances).
              :dataset (make-sampler 10000))))

;;; Return a sampler object that produces MAX-N-SAMPLES number of
;;; random inputs (numbers between 0 and 9).
(defun make-sampler (max-n-samples)
  (make-instance 'function-sampler :max-n-samples max-n-samples
                 :generator (lambda () (random *n-inputs*))))

;;; Log the test error. Also, describe the optimizer and the bpn at
;;; the beginning of training. Called periodically during training
;;; (see above).
(defun log-test-error (optimizer learner)
  (when (zerop (n-instances optimizer))
    (describe optimizer)
    (describe (bpn learner)))
  (log-padded
```

```
   (monitor-bpn-results (make-sampler 1000) (bpn learner)
                        (make-cost-monitors
                         (bpn learner) :attributes `(:event "pred.")))))

#|

;;; Transcript follows:
(repeatably ()
  (let ((*log-time* nil))
    (train-digit-fnn)))
.. training at n-instances: 0
.. train cost: 0.000e+0 (0)
.. #<SEGMENTED-GD-OPTIMIZER {100E112E93}>
..  SEGMENTED-GD-OPTIMIZER description:
..    N-INSTANCES = 0
..    OPTIMIZERS = (#<SGD-OPTIMIZER
..                    #<SEGMENT-SET
..                      (#<->WEIGHT # :SIZE 15 1/1 :NORM 0.04473>
..                       #<->WEIGHT # :SIZE 3 1/1 :NORM 0.01850>
..                       #<->WEIGHT # :SIZE 50 1/1 :NORM 0.07159>
..                       #<->WEIGHT # :SIZE 5 1/1 :NORM 0.03056>)
..                      {100E335B73}>
..                    {100E06DF83}>)
..    SEGMENTS = (#<->WEIGHT (HIDDEN OUTPUT-ACTIVATION) :SIZE
..                  15 1/1 :NORM 0.04473>
..                #<->WEIGHT (:BIAS OUTPUT-ACTIVATION) :SIZE
..                  3 1/1 :NORM 0.01850>
..                #<->WEIGHT (INPUT HIDDEN-ACTIVATION) :SIZE
..                  50 1/1 :NORM 0.07159>
..                #<->WEIGHT (:BIAS HIDDEN-ACTIVATION) :SIZE
..                  5 1/1 :NORM 0.03056>)
..
.. #<SGD-OPTIMIZER {100E06DF83}>
..  GD-OPTIMIZER description:
..    N-INSTANCES = 0
..    SEGMENT-SET = #<SEGMENT-SET
..                    (#<->WEIGHT (HIDDEN OUTPUT-ACTIVATION) :SIZE
..                       15 1/1 :NORM 0.04473>
..                     #<->WEIGHT (:BIAS OUTPUT-ACTIVATION) :SIZE
..                       3 1/1 :NORM 0.01850>
..                     #<->WEIGHT (INPUT HIDDEN-ACTIVATION) :SIZE
..                       50 1/1 :NORM 0.07159>
..                     #<->WEIGHT (:BIAS HIDDEN-ACTIVATION) :SIZE
..                       5 1/1 :NORM 0.03056>)
..                    {100E335B73}>
..    LEARNING-RATE = 1.00000e+0
..    MOMENTUM = 9.00000e-1
..    MOMENTUM-TYPE = :NORMAL
..    WEIGHT-DECAY = 0.00000e+0
..    WEIGHT-PENALTY = 0.00000e+0
..    N-AFTER-UPATE-HOOK = 0
..    BATCH-SIZE = 100
```

```
..
..  BATCH-GD-OPTIMIZER description:
..    N-BEFORE-UPATE-HOOK = 0
..  #<DIGIT-FNN {100E11A423}>
..   BPN description:
..     CLUMPS = #(#<->INPUT INPUT :SIZE 10 1/50 :NORM 0.00000>
..                #<->ACTIVATION
..                  (HIDDEN-ACTIVATION :ACTIVATION) :STRIPES 1/50
..                  :CLUMPS 4>
..                #<->RELU HIDDEN :SIZE 5 1/50 :NORM 0.00000>
..                #<->ACTIVATION
..                  (OUTPUT-ACTIVATION :ACTIVATION) :STRIPES 1/50
..                  :CLUMPS 4>
..                #<->SOFTMAX-XE-LOSS OUTPUT :SIZE 3 1/50 :NORM 0.00000>)
..     N-STRIPES = 1
..     MAX-N-STRIPES = 50
..   pred. cost: 1.100d+0 (1000.00)
.. training at n-instances: 1000
.. train cost: 1.093d+0 (1000.00)
.. training at n-instances: 2000
.. train cost: 5.886d-1 (1000.00)
.. training at n-instances: 3000
.. train cost: 3.574d-3 (1000.00)
.. training at n-instances: 4000
.. train cost: 1.601d-7 (1000.00)
.. training at n-instances: 5000
.. train cost: 1.973d-9 (1000.00)
.. training at n-instances: 6000
.. train cost: 4.882d-10 (1000.00)
.. training at n-instances: 7000
.. train cost: 2.771d-10 (1000.00)
.. training at n-instances: 8000
.. train cost: 2.283d-10 (1000.00)
.. training at n-instances: 9000
.. train cost: 2.123d-10 (1000.00)
.. training at n-instances: 10000
.. train cost: 2.263d-10 (1000.00)
.. pred. cost: 2.210d-10 (1000.00)
..
==> (#<->WEIGHT (:BIAS HIDDEN-ACTIVATION) :SIZE 5 1/1 :NORM 2.94294>
-->  #<->WEIGHT (INPUT HIDDEN-ACTIVATION) :SIZE 50 1/1 :NORM 11.48995>
-->  #<->WEIGHT (:BIAS OUTPUT-ACTIVATION) :SIZE 3 1/1 :NORM 3.39103>
-->  #<->WEIGHT (HIDDEN OUTPUT-ACTIVATION) :SIZE 15 1/1 :NORM 11.39339>)

|#
```

11.3.4 Recurrent Neural Nets

`rnn` Tutorial Hopefully this example from `example/sum-sign-fnn.lisp` illustrates the concepts involved. Make sure you are comfortable with `fnn` Tutorial before reading this.

```
(cl:defpackage :mgl-example-sum-sign-rnn
  (:use #:common-lisp #:mgl))

(in-package :mgl-example-sum-sign-rnn)

;;; There is a single input at each time step...
(defparameter *n-inputs* 1)
;;; and we want to learn the rule that outputs the sign of the sum of
;;; inputs so far in the sequence.
(defparameter *n-outputs* 3)

;;; Generate a training example that's a sequence of random length
;;; between 1 and LENGTH. Elements of the sequence are lists of two
;;; elements:
;;;
;;; 1. The input for the network (a single random number).
;;;
;;; 2. The sign of the sum of inputs so far encoded as 0, 1, 2 (for
;;;    negative, zero and positive values). To add a twist, the sum is
;;;    reset whenever a negative input is seen.
(defun make-sum-sign-instance (&key (length 10))
  (let ((length (max 1 (random length)))
        (sum 0))
    (loop for i below length
          collect (let ((x (1- (* 2 (random 2)))))
                    (incf sum x)
                    (when (< x 0)
                      (setq sum x))
                    (list x (cond ((minusp sum) 0)
                                  ((zerop sum) 1)
                                  (t 2)))))))

;;; Build an RNN with a single lstm hidden layer and softmax output.
;;; For each time step, a SUM-SIGN-FNN will be instantiated.
(defun make-sum-sign-rnn (&key (n-hiddens 1))
  (build-rnn ()
    (build-fnn (:class 'sum-sign-fnn)
      (input (->input :size 1))
      (h (->lstm input :name 'h :size n-hiddens))
      (prediction (->softmax-xe-loss (->activation h :name 'prediction
                                                   :size *n-outputs*))))))

;;; We define this class to be able to specialize how inputs are
;;; translated by adding a SET-INPUT method later.
(defclass sum-sign-fnn (fnn)
  ())

;;; We have a batch of instances from MAKE-SUM-SIGN-INSTANCE for the
;;; RNN. This function is invoked with elements of these instances
;;; belonging to the same time step (i.e. at the same index) and sets
;;; the input and target up.
(defmethod set-input (instances (fnn sum-sign-fnn))
```

```
  (let ((input-nodes (nodes (find-clump 'input fnn))))
    (setf (target (find-clump 'prediction fnn))
          (loop for stripe upfrom 0
                for instance in instances
                collect
                ;; Sequences in the batch are not of equal length. The
                ;; RNN sends a NIL our way if a sequence has run out.
                (when instance
                  (destructuring-bind (input target) instance
                    (setf (mref input-nodes stripe 0) input)
                    target))))))

;;; Train the network by minimizing the loss (cross-entropy here) with
;;; the Adam optimizer.
(defun train-sum-sign-rnn ()
  (let ((rnn (make-sum-sign-rnn)))
    (setf (max-n-stripes rnn) 50)
    ;; Initialize the weights in the usual sqrt(1 / fan-in) style.
    (map-segments (lambda (weights)
                    (let* ((fan-in (mat-dimension (nodes weights) 0))
                           (limit (sqrt (/ 6 fan-in))))
                      (uniform-random! (nodes weights)
                                       :limit (* 2 limit))
                      (.+! (- limit) (nodes weights))))
                  rnn)
    (minimize (monitor-optimization-periodically
               (make-instance 'adam-optimizer
                              :learning-rate 0.2
                              :mean-decay 0.9
                              :mean-decay-decay 0.9
                              :variance-decay 0.9
                              :batch-size 100)
               '((:fn log-test-error :period 30000)
                 (:fn reset-optimization-monitors :period 3000)))
              (make-instance 'bp-learner
                             :bpn rnn
                             :monitors (make-cost-monitors rnn))
              :dataset (make-sampler 30000))))

;;; Return a sampler object that produces MAX-N-SAMPLES number of
;;; random inputs.
(defun make-sampler (max-n-samples &key (length 10))
  (make-instance 'function-sampler :max-n-samples max-n-samples
                 :generator (lambda ()
                              (make-sum-sign-instance :length length))))

;;; Log the test error. Also, describe the optimizer and the bpn at
;;; the beginning of training. Called periodically during training
;;; (see above).
(defun log-test-error (optimizer learner)
  (when (zerop (n-instances optimizer))
    (describe optimizer)
```

```
    (describe (bpn learner)))
  (let ((rnn (bpn learner)))
    (log-padded
     (append
      (monitor-bpn-results (make-sampler 1000) rnn
                           (make-cost-monitors
                            rnn :attributes '(:event "pred.")))
      ;; Same result in a different way: monitor predictions for
      ;; sequences up to length 20, but don't unfold the RNN
      ;; unnecessarily to save memory.
      (let ((*warp-time* t))
        (monitor-bpn-results (make-sampler 1000 :length 20) rnn
                             ;; Just collect and reset the warp
                             ;; monitors after each batch of
                             ;; instances.
                             (make-cost-monitors
                              rnn :attributes '(:event "warped pred."))))))
    ;; Verify that no further unfoldings took place.
    (assert (<= (length (clumps rnn)) 10)))
  (log-mat-room))

#|

;;; Transcript follows:
(let (;; Backprop nets do not need double float. Using single floats
      ;; is faster and needs less memory.
      (*default-mat-ctype* :float)
      ;; Enable moving data in and out of GPU memory so that the RNN
      ;; can work with sequences so long that the unfolded network
      ;; wouldn't otherwise fit in the GPU.
      (*cuda-window-start-time* 1)
      (*log-time* nil))
  ;; Seed the random number generators.
  (repeatably ()
    ;; Enable CUDA if available.
    (with-cuda* ()
      (train-sum-sign-rnn))))
.. training at n-instances: 0
.. cost: 0.000e+0 (0)
.. #<ADAM-OPTIMIZER {1006CD5663}>
..  GD-OPTIMIZER description:
..    N-INSTANCES = 0
..    SEGMENT-SET = #<SEGMENT-SET
..                    (#<->WEIGHT (H #) :SIZE 1 1/1 :NORM 1.73685>
..                     #<->WEIGHT (H #) :SIZE 1 1/1 :NORM 0.31893>
..                     #<->WEIGHT (#1=# #2=# :PEEPHOLE) :SIZE
..                       1 1/1 :NORM 1.81610>
..                     #<->WEIGHT (H #2#) :SIZE 1 1/1 :NORM 0.21965>
..                     #<->WEIGHT (#1# #3=# :PEEPHOLE) :SIZE
..                       1 1/1 :NORM 1.74939>
..                     #<->WEIGHT (H #3#) :SIZE 1 1/1 :NORM 0.40377>
..                     #<->WEIGHT (H PREDICTION) :SIZE
```

```
..                       3 1/1 :NORM 2.15898>
..                      #<->WEIGHT (:BIAS PREDICTION) :SIZE
..                       3 1/1 :NORM 2.94470>
..                      #<->WEIGHT (#1# #4=# :PEEPHOLE) :SIZE
..                       1 1/1 :NORM 0.97601>
..                      #<->WEIGHT (INPUT #4#) :SIZE 1 1/1 :NORM 0.65261>
..                      #<->WEIGHT (:BIAS #4#) :SIZE 1 1/1 :NORM 0.37653>
..                      #<->WEIGHT (INPUT #1#) :SIZE 1 1/1 :NORM 0.92334>
..                      #<->WEIGHT (:BIAS #1#) :SIZE 1 1/1 :NORM 0.01609>
..                      #<->WEIGHT (INPUT #5=#) :SIZE 1 1/1 :NORM 1.09995>
..                      #<->WEIGHT (:BIAS #5#) :SIZE 1 1/1 :NORM 1.41244>
..                      #<->WEIGHT (INPUT #6=#) :SIZE 1 1/1 :NORM 0.40475>
..                      #<->WEIGHT (:BIAS #6#) :SIZE 1 1/1 :NORM 1.75358>)
..                     {1006CD8753}>
..    LEARNING-RATE = 2.00000e-1
..    MOMENTUM = NONE
..    MOMENTUM-TYPE = :NONE
..    WEIGHT-DECAY = 0.00000e+0
..    WEIGHT-PENALTY = 0.00000e+0
..    N-AFTER-UPATE-HOOK = 0
..    BATCH-SIZE = 100
..
..  BATCH-GD-OPTIMIZER description:
..    N-BEFORE-UPATE-HOOK = 0
..
..  ADAM-OPTIMIZER description:
..    MEAN-DECAY-RATE = 1.00000e-1
..    MEAN-DECAY-RATE-DECAY = 9.00000e-1
..    VARIANCE-DECAY-RATE = 1.00000e-1
..    VARIANCE-ADJUSTMENT = 1.00000d-7
..  #<RNN {10047C77E3}>
..   BPN description:
..     CLUMPS = #(#<SUM-SIGN-FNN :STRIPES 1/50 :CLUMPS 4>
..                #<SUM-SIGN-FNN :STRIPES 1/50 :CLUMPS 4>)
..     N-STRIPES = 1
..     MAX-N-STRIPES = 50
..
..   RNN description:
..     MAX-LAG = 1
..   pred.        cost: 1.223e+0 (4455.00)
.. warped pred. cost: 1.228e+0 (9476.00)
.. Foreign memory usage:
.. foreign arrays: 162 (used bytes: 39,600)
.. CUDA memory usage:
.. device arrays: 114 (used bytes: 220,892, pooled bytes: 19,200)
.. host arrays: 162 (used bytes: 39,600)
.. host->device copies: 6,164, device->host copies: 4,490
.. training at n-instances: 3000
.. cost: 3.323e-1 (13726.00)
.. training at n-instances: 6000
.. cost: 3.735e-2 (13890.00)
.. training at n-instances: 9000
```

```
.. cost: 1.012e-2 (13872.00)
.. training at n-instances: 12000
.. cost: 3.026e-3 (13953.00)
.. training at n-instances: 15000
.. cost: 9.267e-4 (13948.00)
.. training at n-instances: 18000
.. cost: 2.865e-4 (13849.00)
.. training at n-instances: 21000
.. cost: 8.893e-5 (13758.00)
.. training at n-instances: 24000
.. cost: 2.770e-5 (13908.00)
.. training at n-instances: 27000
.. cost: 8.514e-6 (13570.00)
.. training at n-instances: 30000
.. cost: 2.705e-6 (13721.00)
.. pred.        cost: 1.426e-6 (4593.00)
.. warped pred. cost: 1.406e-6 (9717.00)
.. Foreign memory usage:
.. foreign arrays: 216 (used bytes: 52,800)
.. CUDA memory usage:
.. device arrays: 148 (used bytes: 224,428, pooled bytes: 19,200)
.. host arrays: 216 (used bytes: 52,800)
.. host->device copies: 465,818, device->host copies: 371,990
..
==> (#<->WEIGHT (H (H :OUTPUT)) :SIZE 1 1/1 :NORM 0.10624>
-->  #<->WEIGHT (H (H :CELL)) :SIZE 1 1/1 :NORM 0.94460>
-->  #<->WEIGHT ((H :CELL) (H :FORGET) :PEEPHOLE) :SIZE 1 1/1 :NORM 0.61312>
-->  #<->WEIGHT (H (H :FORGET)) :SIZE 1 1/1 :NORM 0.38093>
-->  #<->WEIGHT ((H :CELL) (H :INPUT) :PEEPHOLE) :SIZE 1 1/1 :NORM 1.17956>
-->  #<->WEIGHT (H (H :INPUT)) :SIZE 1 1/1 :NORM 0.88011>
-->  #<->WEIGHT (H PREDICTION) :SIZE 3 1/1 :NORM 49.93808>
-->  #<->WEIGHT (:BIAS PREDICTION) :SIZE 3 1/1 :NORM 10.98112>
-->  #<->WEIGHT ((H :CELL) (H :OUTPUT) :PEEPHOLE) :SIZE 1 1/1 :NORM 0.67996>
-->  #<->WEIGHT (INPUT (H :OUTPUT)) :SIZE 1 1/1 :NORM 0.65251>
-->  #<->WEIGHT (:BIAS (H :OUTPUT)) :SIZE 1 1/1 :NORM 10.23003>
-->  #<->WEIGHT (INPUT (H :CELL)) :SIZE 1 1/1 :NORM 5.98116>
-->  #<->WEIGHT (:BIAS (H :CELL)) :SIZE 1 1/1 :NORM 0.10681>
-->  #<->WEIGHT (INPUT (H :FORGET)) :SIZE 1 1/1 :NORM 4.46301>
-->  #<->WEIGHT (:BIAS (H :FORGET)) :SIZE 1 1/1 :NORM 1.57195>
-->  #<->WEIGHT (INPUT (H :INPUT)) :SIZE 1 1/1 :NORM 0.36401>
-->  #<->WEIGHT (:BIAS (H :INPUT)) :SIZE 1 1/1 :NORM 8.63833>)

|#
```

- **[class]** **rnn** *bpn*

  A recurrent neural net (as opposed to a feed-forward one. It is typically built with `build-rnn` that's no more than a shallow convenience macro.

  An `rnn` takes instances as inputs that are sequences of variable length. At each time step, the next unprocessed elements of these sequences are set as input until all input sequences in the batch run out. To be able to perform backpropagation, all intermediate `lump`s must

be kept around, so the recursive connections are transformed out by unfolding the network. Just how many lumps this means depends on the length of the sequences.

When an `rnn` is created, `max-lag + 1` `bpn`s are instantiated so that all weights are present and one can start training it.

- **[reader]** **unfolder** *rnn (:unfolder)*

  The `unfolder` of an `rnn` is function of no arguments that builds and returns a `bpn`. The unfolder is allowed to create networks with arbitrary topology even different ones for different `time-step`s with the help of `lag`, or nested `rnn`s. Weights of the same name are shared between the folds. That is, if a `->weight` lump were to be created and a weight lump of the same name already exists, then the existing lump will be added to the `bpn` created by `unfolder`.

- **[reader]** **max-lag** *rnn (:max-lag = 1)*

  The networks built by `unfolder` may contain new weights up to time step `max-lag`. Beyond that point, all weight lumps must be reappearances of weight lumps with the same name at previous time steps. Most recurrent networks reference only the state of lumps at the previous time step (with the function `lag`), hence the default of 1. But it is possible to have connections to arbitrary time steps. The maximum connection lag must be specified when creating the `rnn`.

- **[accessor]** **cuda-window-start-time** *rnn (:cuda-window-start-time = *cuda-window-start-time*)*

  Due to unfolding, the memory footprint of an `rnn` is almost linear in the number of time steps (i.e. the max sequence length). For prediction, this is addressed by Time Warp. For training, we cannot discard results of previous time steps because they are needed for backpropagation, but we can at least move them out of GPU memory if they are not going to be used for a while and copy them back before they are needed. Obviously, this is only relevant if CUDA is being used.

  If `cuda-window-start-time` is `nil`, then this feature is turned off. Else, during training, at `cuda-window-start-time` or later time steps, matrices belonging to non-weight lumps may be forced out of GPU memory and later brought back as neeeded.

  This feature is implemented in terms of `mgl-mat:with-syncing-cuda-facets` that uses CUDA host memory (also known as *page-locked* or *pinned memory*) to do asynchronous copies concurrently with normal computation. The consequence of this is that it is now main memory usage that's unbounded which toghether with page-locking makes it a potent weapon to bring a machine to a halt. You were warned.

- **[variable]** ***cuda-window-start-time*** *nil*

  The default for `cuda-window-start-time`.

- **[macro]** **build-rnn** *(&key rnn (class 'rnn) name initargs max-n-stripes (max-lag 1)) &body body*

  Create an `rnn` with `max-n-stripes` and `max-lag` whose `unfolder` is `body` wrapped in a lambda. Bind symbol given as the `rnn` argument to the `rnn` object so that `body` can see it.

- **[function]** **lag** *name &key (lag 1) rnn path*

  In `rnn` or if it's `nil` the `rnn` being extended with another `bpn` (called *unfolding*), look up the `clump` with `name` in the `bpn` that's `lag` number of time steps before the `bpn` being added. If this function is called from `unfolder` of an `rnn` (which is what happens behind the scene in the body of `build-rnn`), then it returns an opaque object representing a lagged connection to a clump, else it returns the `clump` itself.

  FIXDOC: `path`

- **[function]** **time-step** *&key (rnn *rnn*)*

  Return the time step `rnn` is currently executing or being unfolded for. It is 0 when the `rnn` is being unfolded for the first time.

- **[method]** **set-input** *instances (rnn rnn)*

  `rnn`s operate on batches of instances just like `fnn`s. But the instances here are like datasets: sequences or samplers and they are turned into sequences of batches of instances with `map-datasets` `:impute` `nil`. The batch of instances at index 2 is clamped onto the `bpn` at time step 2 with `set-input`.

  When the input sequences in the batch are not of the same length, already exhausted sequences will produce `nil` (due to `:impute` `nil`) above. When such a `nil` is clamped with `set-input` on a `bpn` of the `rnn`, `set-input` must set the `importance` of the ->ERROR lumps to 0 else training would operate on the noise left there by previous invocations.

Time Warp The unbounded memory usage of `rnn`s with one `bpn` allocated per time step can become a problem. For training, where the gradients often have to be backpropagated from the last time step to the very beginning, this is hard to solve but with `cuda-window-start-time` the limit is no longer GPU memory.

For prediction on the other hand, one doesn't need to keep old steps around indefinitely: they can be discarded when future time steps will never reference them again.

- **[variable]** ***warp-time*** *nil*

  Controls whether warping is enabled (see Time Warp). Don't enable it for training, as it would make backprop impossible.

- **[function]** **warped-time** *&key (rnn *rnn*) (time (time-step :rnn rnn)) (lag 0)*

  Return the index of the `bpn` in `clumps` of `rnn` whose task it is to execute computation at `(- (time-step rnn) lag)`. This is normally the same as `time-step` (disregarding `lag`). That is, `clumps` can be indexed by `time-step` to get the `bpn`. However, when `*warp-time*` is true, execution proceeds in a cycle as the structure of the network allows.

  Suppose we have a typical `rnn` that only ever references the previous time step so its `max-lag` is 1. Its `unfolder` returns `bpn`s of identical structure bar a shift in their time lagged connections except for the very first, so `warp-start` and `warp-length` are both 1. If `*warp-time*` is `nil`, then the mapping from `time-step` to the `bpn` in `clumps` is straightforward:

```
time:   |  0 |  1 |  2 |  3 |  4 |  5
--------+----+----+----+----+----+----
warped: |  0 |  1 |  2 |  3 |  4 |  5
--------+----+----+----+----+----+----
bpn:    | b0 | b1 | b2 | b3 | b4 | b5
```

When *warp-time* is true, we reuse the b1 - b2 bpns in a loop:

```
time:   |  0 |  1 |  2 |  3 |  4 |  5
--------+----+----+----+----+----+----
warped: |  0 |  1 |  2 |  1 |  2 |  1
--------+----+----+----+----+----+----
bpn:    | b0 | b1 | b2 | b1*| b2 | b1*
```

b1* is the same bpn as b1, but its connections created by lag go through warped time and end up referencing b2. This way, memory consumption is independent of the number time steps needed to process a sequence or make predictions.

To be able to pull this trick off warp-start and warp-length must be specified when the rnn is instantiated. In general, with *warp-time* (+ warp-start (max 2 warp-length)) bpns are needed. The 2 comes from the fact that with cycle length 1 a bpn would need to takes its input from itself which is problematic because it has nodes for only one set of values.

- **[reader]** **warp-start** *rnn (:warp-start = 1)*

  The time-step from which unfolder will create bpns that essentially repeat every warp-length steps.

- **[reader]** **warp-length** *rnn (:warp-length = 1)*

  An integer such that the bpn unfolder creates at time step i (where (<= warp-start i)) is identical to the bpn created at time step (+ warp-start (mod (- i warp-start) warp-length)) except for a shift in its time lagged connections.

- **[accessor]** **step-monitors** *rnn (:step-monitors = nil)*

  During training, unfolded bpns corresponding to previous time steps may be expensive to get at because they are no longer in GPU memory. This consideration also applies to making prediction with the additional caveat that with *warp-time* true, previous states are discarded so it's not possible to gather statistics after forward finished.

  Add monitor objects to this slot and they will be automatically applied to the rnn after each step when forwarding the rnn during training or prediction. To be able to easily switch between sets of monitors, in addition to a list of monitors this can be a symbol or a function, too. If it's a symbol, then its a designator for its symbol-value. If it's a function, then it must have no arguments and it's a designator for its return value.

11.4 Lumps

11.4.1 Lump Base Class

- **[class]** **lump** *clump*

  A `lump` is a simple, layerlike component of a neural network. There are many kinds of lumps, each of which performs a specific operation or just stores inputs and weights. By convention, the names of lumps start with the prefix `->`. Defined as classes, they also have a function of the same name as the class to create them easily. These maker functions typically have keyword arguments corresponding to initargs of the class, with some (mainly the input lumps) turned into normal positional arguments. So instead of having to do

  ```
  (make-instance '->tanh :x some-input :name 'my-tanh)
  ```

  one can simply write

  ```
  (->tanh some-input :name 'my-tanh)
  ```

  Lumps instantiated in any way within a `build-fnn` or `build-rnn` are automatically added to the network being built.

  A lump has its own `nodes` and `derivatives` matrices allocated for it in which the results of the forward and backward passes are stored. This is in contrast to a `bpn` whose `nodes` and `derivatives` are those of its last constituent `clump`.

  Since lumps almost always live within a `bpn`, their `n-stripes` and `max-n-stripes` are handled automagically behind the scenes.

- **[reader]** **size** *lump (:size)*

  The number of values in a single stripe.

- **[reader]** **default-value** *lump (:default-value = 0)*

  Upon creation or resize the lump's nodes get filled with this value.

- **[generic-function]** **default-size** *lump*

  Return a default for the `size` of `lump` if one is not supplied at instantiation. The value is often computed based on the sizes of the inputs. This function is for implementing new lump types.

- **[reader]** **nodes** *lump (= nil)*

  The values computed by the lump in the forward pass are stored here. It is an `n-stripes * size` matrix that has storage allocated for `max-n-stripes * size` elements for non-weight lumps. `->weight` lumps have no stripes nor restrictions on their shape.

- **[reader]** **derivatives** *lump*

  The derivatives computed in the backward pass are stored here. This matrix is very much like `nodes` in shape and size.

11.4.2 Inputs

Input Lump

- **[class]** **->input** *->dropout*

  A lump that has no input lumps, does not change its values in the forward pass (except when `dropout` is non-zero), and does not compute derivatives. *Clamp* inputs on `nodes` of input lumps in `set-input`.

  For convenience, `->input` can perform dropout itself although it defaults to no dropout.

```
(->input :size 10 :name 'some-input)
==> #<->INPUT SOME-INPUT :SIZE 10 1/1 :NORM 0.00000>
```

- **[accessor]** **dropout** *->input (= nil)*

  See `dropout`.

Embedding Lump This lump is like an input and a simple activation molded together in the name of efficiency.

- **[class]** **->embedding** *lump*

  Select rows of `weights`, one row for each index in `input-row-indices`. This lump is equivalent to adding an `->input` lump with a one hot encoding scheme and a `->v*m` lump on top of it, but it is more efficient in execution and in memory usage, because it works with a sparse representation of the input.

  The `size` of this lump is the number of columns of `weights` which is determined automatically.

```
(->embedding :weights (->weight :name 'embedding-weights
                                :dimensions '(3 5))
             :name 'embeddings)
==> #<->EMBEDDING EMBEDDINGS :SIZE 5 1/1 :NORM 0.00000>
```

- **[reader]** **weights** *->embedding (:weights)*

  A weight lump whose rows indexed by `input-row-indices` are copied to the output of this lump.

- **[accessor]** **input-row-indices** *->embedding (:input-row-indices)*

  A sequence of batch size length of row indices. To be set in `set-input`.

11.4.3 Weight Lump

- **[class]** **->weight** *lump*

  A set of optimizable parameters of some kind. When a `bpn` is is trained (see Training) the `nodes` of weight lumps will be changed. Weight lumps perform no computation.

  Weights can be created by specifying the total size or the dimensions:

```
(dimensions (->weight :size 10 :name 'w))
=> (1 10)
(dimensions (->weight :dimensions '(5 10) :name 'w))
=> (5 10)
```

- **[reader]** **dimensions** *->weight (:dimensions)*

  `nodes` and `derivatives` of this lump will be allocated with these dimensions.

- **[macro]** **with-weights-copied** *(from-bpn) &body body*

  In `body` `->weight` will first look up if a weight lump of the same name exists in `from-bpn` and return that, or else create a weight lump normally. If `from-bpn` is `nil`, then no weights are copied.

11.4.4 Activations

Activation Subnet So we have some inputs. Usually the next step is to multiply the input vector with a weight matrix and add biases. This can be done directly with `->+`, `->v*m` and `->weight`, but it's more convenient to use activation subnets to reduce the clutter.

- **[class]** **->activation** *bpn*

  Activation subnetworks are built by the function `->activation` and they have a number of lumps hidden inside them. Ultimately, this subnetwork computes a sum like `sum_i x_i * W_i + sum_j y_j .* V_j + biases` where `x_i` are input lumps, `W_i` are dense matrices representing connections, while `V_j` are peephole connection vectors that are mulitplied in an elementwise manner with their corresponding input `y_j`.

- **[function]** **->activation** *inputs &key (name (gensym)) size peepholes (add-bias-p t)*

  Create a subnetwork of class `->activation` that computes the over activation from dense connection from lumps in `inputs`, and elementwise connection from lumps in `peepholes`. Create new `->weight` lumps as necessary. `inputs` and `peepholes` can be a single lump or a list of lumps. Finally, if `add-bias-p`, then add an elementwise bias too. `size` must be specified explicitly, because it is not possible to determine it unless there are peephole connections.

  ```
  (->activation (->input :size 10 :name 'input) :name 'h1 :size 4)
  ==> #<->ACTIVATION (H1 :ACTIVATION) :STRIPES 1/1 :CLUMPS 4>
  ```

  This is the basic workhorse of neural networks which takes care of the linear transformation whose results and then fed to some non-linearity (`->sigmoid`, `->tanh`, etc).

  The name of the subnetwork clump is `(,name :activation)`. The bias weight lump (if any) is named `(:bias ,name)`. Dense connection weight lumps are named are named after the input and `name`: `(,(name input) ,name)`, while peepholes weight lumps are named `(,(name input) ,name :peephole)`. This is useful to know if, for example, they are to be initialized differently.

Batch-Normalization

- **[class]** **->batch-normalized** *lump*

  This is an implementation of v3 of the Batch Normalization paper. The output of `->batch-normalized` is its input normalized so that for all elements the mean across stripes is zero and the variance is 1. That is, the mean of the batch is subtracted from the inputs and they are rescaled by their sample stddev. Actually, after the normalization step the values are rescaled and shifted (but this time with learnt parameters) in order to keep the representational power of the model the same. The primary purpose of this lump is to speed up learning, but it also acts as a regularizer. See the paper for the details.

  To normalize the output of `lump` without no additional regularizer effect:

  ```
  (->batch-normalized lump :batch-size :use-population)
  ```

  The above uses an exponential moving average to estimate the mean and variance of batches and these estimations are used at both training and test time. In contrast to this, the published version uses the sample mean and variance of the current batch at training time which injects noise into the process. The noise is higher for lower batch sizes and has a regularizing effect. This is the default behavior (equivalent to `:batch-size nil`):

  ```
  (->batch-normalized lump)
  ```

  For performance reasons one may wish to process a higher number of instances in a batch (in the sense of `n-stripes`) and get the regularization effect associated with a lower batch size. This is possible by setting `:batch-size` to a divisor of the the number of stripes. Say, the number of stripes is 128, but we want as much regularization as we would get with 32:

  ```
  (->batch-normalized lump :batch-size 32)
  ```

  The primary input of `->batch-normalized` is often an `->activation`(`0` `1`) and its output is fed into an activation function (see Activation Functions).

- **[reader]** **batch-normalization** *->batch-normalized (:normalization)*

  The `->batch-normalization` of this lump. May be shared between multiple `->batch-normalized` lumps.

  Batch normalization is special in that it has state apart from the computed results (`nodes`) and its derivatives (`derivatives`). This state is the estimated mean and variance of its inputs and they are encapsulated by `->batch-normalization`.

  If `normalization` is not given at instantiation, then a new `->batch-normalization` object will be created automatically, passing `:batch-size`, `:variance-adjustment`, and `:population-decay` arguments on to `->batch-normalization`. See `batch-size`, `variance-adjustment` and `population-decay`. New scale and shift weight lumps will be created with names:

  ```
  `(,name :scale)
  `(,name :shift)
  ```

  where `name` is the `name` of this lump.

This default behavior covers the use-case where the statistics kept by `->batch-normalization` are to be shared only between time steps of an `rnn`.

- **[class]** **->batch-normalization** *->weight*

  The primary purpose of this class is to hold the estimated mean and variance of the inputs to be normalized and allow them to be shared between multiple `->batch-normalized` lumps that carry out the computation. These estimations are saved and loaded by `save-state` and `load-state`.

```
(->batch-normalization (->weight :name '(h1 :scale) :size 10)
                       (->weight :name '(h1 :shift) :size 10)
                       :name '(h1 :batch-normalization))
```

- **[reader]** **scale** *->batch-normalization (:scale)*

  A weight lump of the same size as `shift`. This is γ in the paper.

- **[reader]** **shift** *->batch-normalization (:shift)*

  A weight lump of the same size as `scale`. This is β in the paper.

- **[reader]** **batch-size** *->batch-normalization (:batch-size = nil)*

  Normally all stripes participate in the batch. Lowering the number of stripes may increase the regularization effect, but it also makes the computation less efficient. By setting `batch-size` to a divisor of `n-stripes` one can decouple the concern of efficiency from that of regularization. The default value, `nil`, is equivalent to `n-stripes`. `batch-size` only affects training.

  With the special value `:use-population`, instead of the mean and the variance of the current batch, use the population statistics for normalization. This effectively cancels the regularization effect, leaving only the faster learning.

- **[reader]** **variance-adjustment** *->batch-normalization (:variance-adjustment = 1.0e-4)*

  A small positive real number that's added to the sample variance. This is ϵ in the paper.

- **[reader]** **population-decay** *->batch-normalization (:population-decay = 0.99)*

  While training, an exponential moving average of batch means and standard deviances (termed *population statistics*) is updated. When making predictions, normalization is performed using these statistics. These population statistics are persisted by `save-state`.

- **[function]** **->batch-normalized-activation** *inputs &key (name (gensym)) size peepholes batch-size variance-adjustment population-decay*

  A utility functions that creates and wraps an `->activation`(`0` `1`) in `->batch-normalized` and with its `batch-normalization` the two weight lumps for the scale and shift parameters. `(->batch-normalized-activation inputs :name 'h1 :size 10)` is equivalent to:

```
(->batch-normalized (->activation inputs :name 'h1 :size 10 :add-bias-p nil)
                    :name '(h1 :batch-normalized-activation))
```

Note how biases are turned off since normalization will cancel them anyway (but a shift is added which amounts to the same effect).

11.4.5 Activation Functions

Now we are moving on to the most important non-linearities to which activations are fed.

Sigmoid Lump

- **[class]** **->sigmoid** *->dropout lump*

  Applies the `1/(1 + e^{-x})` function elementwise to its inputs. This is one of the classic non-linearities for neural networks.

  For convenience, `->sigmoid` can perform dropout itself although it defaults to no dropout.

```
(->sigmoid (->activation (->input :size 10) :size 5) :name 'this)
==> #<->SIGMOID THIS :SIZE 5 1/1 :NORM 0.00000>
```

  The `size` of this lump is the size of its input which is determined automatically.

- **[accessor]** **dropout** *->sigmoid (= nil)*

  See `dropout`.

Tanh Lump

- **[class]** **->tanh** *lump*

  Applies the `tanh` function to its input in an elementwise manner. The `size` of this lump is the size of its input which is determined automatically.

Scaled Tanh Lump

- **[class]** **->scaled-tanh** *lump*

  Pretty much like `tanh` but its input and output is scaled in such a way that the variance of its output is close to 1 if the variance of its input is close to 1 which is a nice property to combat vanishing gradients. The actual function is `1.7159 * tanh(2/3 * x)`. The `size` of this lump is the size of its input which is determined automatically.

Relu Lump We are somewhere around year 2007 by now.

- **[class]** **->relu** *lump*

  `max(0,x)` activation function. Be careful, relu units can get stuck in the off state: if they move to far to negative territory it can be very difficult to get out of it. The `size` of this lump is the size of its input which is determined automatically.

Max Lump We are in about year 2011.

- **[class]** **->max** *lump*

  This is basically maxout without dropout (see http://arxiv.org/abs/1302.4389). It groups its inputs by `group-size`, and outputs the maximum of each group. The `size` of the output is automatically calculated, it is the size of the input divided by `group-size`.

```
(->max (->input :size 120) :group-size 3 :name 'my-max)
==> #<->MAX MY-MAX :SIZE 40 1/1 :NORM 0.00000 :GROUP-SIZE 3>
```

  The advantage of `->max` over `->relu` is that flow gradient is never stopped so there is no problem of units getting stuck in off state.

- **[reader]** **group-size** *->max (:group-size)*

  The number of inputs in each group.

Min Lump

- **[class]** **->min** *lump*

  Same as `->max`, but it computes the `min` of groups. Rarely useful.

- **[reader]** **group-size** *->min (:group-size)*

  The number of inputs in each group.

Max-Channel Lump

- **[class]** **->max-channel** *lump*

  Called LWTA (Local Winner Take All) or Channel-Out (see http://arxiv.org/abs/1312.1909) in the literature it is basically `->max`, but instead of producing one output per group, it just produces zeros for all unit but the one with the maximum value in the group. This allows the next layer to get some information about the path along which information flowed. The `size` of this lump is the size of its input which is determined automatically.

- **[reader]** **group-size** *->max-channel (:group-size)*

  The number of inputs in each group.

11.4.6 Losses

Ultimately, we need to tell the network what to learn which means that the loss function to be minimized needs to be constructed as part of the network.

Loss Lump

- **[class]** **->loss** *->sum*

  Calculate the loss for the instances in the batch. The main purpose of this lump is to provide a training signal.

An error lump is usually a leaf in the graph of lumps (i.e. there are no other lumps whose input is this one). The special thing about error lumps is that 1 (but see `importance`) is added automatically to their derivatives. Error lumps have exactly one node (per stripe) whose value is computed as the sum of nodes in their input lump.

- **[accessor]** **importance** *->loss (:importance = nil)*

  This is to support weighted instances. That is when not all training instances are equally important. If non-`nil`, a 1d `mat` with the importances of stripes of the batch. When `importance` is given (typically in `set-input`), then instead of adding 1 to the derivatives of all stripes, `importance` is added elemtwise.

Squared Difference Lump In regression, the squared error loss is most common. The squared error loss can be constructed by combining `->squared-difference` with a `->loss`.

- **[class]** **->squared-difference** *lump*

  This lump takes two input lumps and calculates their squared difference `(x - y)^2` in an elementwise manner. The `size` of this lump is automatically determined from the size of its inputs. This lump is often fed into `->loss` that sums the squared differences and makes it part of the function to be minimized.

```
(->loss (->squared-difference (->activation (->input :size 100)
                                            :size 10)
                              (->input :name 'target :size 10))
        :name 'squared-error)
==> #<->LOSS SQUARED-ERROR :SIZE 1 1/1 :NORM 0.00000>
```

  Currently this lump is not CUDAized, but it will copy data from the GPU if it needs to.

Softmax Cross-Entropy Loss Lump

- **[class]** **->softmax-xe-loss** *lump*

  A specialized lump that computes the softmax of its input in the forward pass and backpropagates a cross-entropy loss. The advantage of doing these together is numerical stability. The total cross-entropy is the sum of cross-entropies per group of `group-size` elements:

$$XE(x) = -\sum_{i=1,g} t_i \ln(s_i),$$

  where `g` is the number of classes (`group-size`), `t_i` are the targets (i.e. the true probabilities of the class, often all zero but one), `s_i` is the output of softmax calculated from input `x`:

$$s_i = softmax(x_1, x_2, ..., x_g) = \frac{e_i^x}{\sum_{j=1,g} e_j^x}$$

  In other words, in the forward phase this lump takes input `x`, computes its elementwise `exp`, normalizes each group of `group-size` elements to sum to 1 to get the softmax which is

the result that goes into `nodes`. In the backward phase, there are two sources of gradients: the lumps that use the output of this lump as their input (currently not implemented and would result in an error) and an implicit cross-entropy loss.

One can get the cross-entropy calculated in the most recent forward pass by calling `cost` on this lump.

This is the most common loss function for classification. In fact, it is nearly ubiquitous. See the `fnn` Tutorial and the `rnn` Tutorial for how this loss and `set-input` work together.

- **[reader]** **group-size** *->softmax-xe-loss (:group-size)*

  The number of elements in a softmax group. This is the number of classes for classification. Often `group-size` is equal to `size` (it is the default), but in general the only constraint is that `size` is a multiple of `group-size`.

- **[accessor]** **target** *->softmax-xe-loss (:target = nil)*

  Set in `set-input`, this is either a `mat` of the same size as the input lump `x` or if the target is very sparse, this can also be a sequence of batch size length that contains the index value pairs of non-zero entries:

```
(;; first instance in batch has two non-zero targets
 (;; class 10 has 30% expected probability
  (10 . 0.3)
  ;; class 2 has 70% expected probability
  (2 .  0.7))
 ;; second instance in batch puts 100% on class 7
 7
 ;; more instances in the batch follow
 ...)
```

  Actually, in the rare case where `group-size` is not `size` (i.e. there are several softmax normalization groups for every example), the length of the above target sequence is `batch-size` * N-GROUPS. Indices are always relative to the start of the group.

  If `group-size` is large (for example, in neural language models with a huge number of words), using sparse targets can make things go much faster, because calculation of the derivative is no longer quadratic.

  Giving different weights to training instances is implicitly supported. While target values in a group should sum to 1, multiplying all target values with a weight `w` is equivalent to training that `w` times on the same example.

- **[function]** **ensure-softmax-target-matrix** *softmax-xe-loss n*

  Set `target` of `softmax-xe-loss` to a `mat` capable of holding the dense target values for `n` stripes.

11.4.7 Stochasticity

Dropout Lump

- **[class]** **->dropout** *lump*

  The output of this lump is identical to its input, except it randomly zeroes out some of them during training which act as a very strong regularizer. See Geoffrey Hinton's 'Improving neural networks by preventing co-adaptation of feature detectors'.

  The `size` of this lump is the size of its input which is determined automatically.

- **[accessor]** **dropout** *->dropout (:dropout = 0.5)*

  If non-`nil`, then in the forward pass zero out each node in this chunk with `dropout` probability.

Gaussian Random Lump

- **[class]** **->gaussian-random** *lump*

  This lump has no input, it produces normally distributed independent random numbers with `mean` and `variance` (or `variance-for-prediction`). This is useful building block for noise based regularization methods.

```
(->gaussian-random :size 10 :name 'normal :mean 1 :variance 2)
==> #<->GAUSSIAN-RANDOM NORMAL :SIZE 10 1/1 :NORM 0.00000>
```

- **[accessor]** **mean** *->gaussian-random (:mean = 0)*

  The mean of the normal distribution.

- **[accessor]** **variance** *->gaussian-random (:variance = 1)*

  The variance of the normal distribution.

- **[accessor]** **variance-for-prediction** *->gaussian-random (:variance-for-prediction = 0)*

  If not `nil`, then this value overrides `variance` when not in training (i.e. when making predictions).

Binary Sampling Lump

- **[class]** **->sample-binary** *lump*

  Treating values of its input as probabilities, sample independent binomials. Turn true into 1 and false into 0. The `size` of this lump is determined automatically from the size of its input.

```
(->sample-binary (->input :size 10) :name 'binarized-input)
==> #<->SAMPLE-BINARY BINARIZED-INPUT :SIZE 10 1/1 :NORM 0.00000>
```

11.4.8 Arithmetic

Sum Lump

- **[class]** **->sum** *lump*

  Computes the sum of all nodes of its input per stripe. This `size` of this lump is always 1.

Vector-Matrix Multiplication Lump

- **[class]** **->v*m** *lump*

  Perform `x * weights` where `x` (the input) is of size `m` and `weights` is a `->weight` whose single stripe is taken to be of dimensions `M x N` stored in row major order. `n` is the size of this lump. If `transpose-weights-p` then `weights` is `N x M` and `x * weights'` is computed.

- **[reader]** **weights** *->v*m (:weights)*

  A `->weight` lump.

- **[reader]** **transpose-weights-p** *->v*m (:transpose-weights-p = nil)*

  Determines whether the input is multiplied by `weights` or its transpose.

Elementwise Addition Lump

- **[class]** **->+** *lump*

  Performs elementwise addition on its input lumps. The `size` of this lump is automatically determined from the size of its inputs if there is at least one. If one of the inputs is a `->weight` lump, then it is added to every stripe.

```
(->+ (list (->input :size 10) (->weight :size 10 :name 'bias))
     :name 'plus)
==> #<->+ PLUS :SIZE 10 1/1 :NORM 0.00000>
```

Elementwise Multiplication Lump

- **[class]** **->*** *lump*

  Performs elementwise multiplication on its two input lumps. The `size` of this lump is automatically determined from the size of its inputs. Either input can be a `->weight` lump.

```
(->* (->input :size 10) (->weight :size 10 :name 'scale)
     :name 'mult)
==> #<->* MULT :SIZE 10 1/1 :NORM 0.00000>
```

Abs Lump

- **[class]** **->abs** *lump*

Exp Lump

- **[class]** **->exp** *lump*

Normalized Lump

- **[class]** **->normalized** *lump*

Sine Lump

- **[class]** **->sin** *lump*

  Applies the `sin` function to its input in an elementwise manner. The `size` of this lump is the size of its input which is determined automatically.

11.4.9 Operations for `rnns`

LSTM Subnet

- **[class]** **->lstm** *bpn*

  Long-Short Term Memory subnetworks are built by the function `->lstm` and they have many lumps hidden inside them. These lumps are packaged into a subnetwork to reduce clutter.

- **[function]** **->lstm** *inputs &key name cell-init output-init size (activation-fn '->activation) (gate-fn '->sigmoid) (input-fn '->tanh) (output-fn '->tanh) (peepholes t)*

  Create an LSTM layer consisting of input, forget, output gates with which input, cell state and output are scaled. Lots of lumps are created, the final one representing to output of the LSTM has `name`. The rest of the lumps are named automatically based on `name`. This function returns only the output lump (`m`), but all created lumps are added automatically to the `bpn` being built.

  There are many papers and tutorials on LSTMs. This version is well described in "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling" (2014, Hasim Sak, Andrew Senior, Francoise Beaufays). Using the notation from that paper:

$$i_t = s(W\_ixx\_t + W\_imm\_t - 1 + W\_ic \odot c\_t - 1 + b\_i)$$

$$f\_t = s(W\_fxx\_t + W\_fmm\_t - 1 + W\_fc \odot c\_t - 1 + b\_f)$$

$$c\_t = f\_t \odot c\_t - 1 + i\_t \odot g(W\_cxx\_t + W\_cmm\_t - 1 + b\_c)$$

$$o\_t = s(W\_oxx\_t + W\_omm\_t - 1 + W\_oc \odot c\_t + b\_o)$$

$$m\_t = o\_t \odot h(c\_t),$$

  where `i`, `f`, and `o` are the input, forget and output gates. `c` is the cell state and `m` is the actual output.

Weight matrices for connections from `c` (`W_ic`, `W_fc` and `W_oc`) are diagonal and represented by just the vector of diagonal values. These connections are only added if `peepholes` is true.

A notable difference from the paper is that in addition to being a single lump, `x_t` (`inputs`) can also be a list of lumps. Whenever some activation is to be calculated based on `x_t`, it is going to be the sum of individual activations. For example, `W_ix * x_t` is really `sum_j W_ijx * inputs_j`.

If `cell-init` is non-`nil`, then it must be a `clump` of `size` form which stands for the initial state of the value cell (`c_{-1}`). `cell-init` being `nil` is equivalent to the state of all zeros.

`activation-fn` defaults to `->activation(0 1)`, but it can be for example `->batch-normalized-activation`. In general, functions like the aforementioned two with signature like (`inputs &key name size peepholes`) can be passed as `activation-fn`.

Sequence Barrier Lump

- **[class]** **->seq-barrier** *lump*

  In an `rnn`, processing of stripes (instances in the batch) may require different number of time step so the final state for stripe 0 is in stripe 0 of some lump L at time step 7, while for stripe 1 it is in stripe 1 of sump lump L at time step 42.

  This lump copies the per-stripe states from different lumps into a single lump so that further processing can take place (typically when the `rnn` is embedded in another network).

  The `size` of this lump is automatically set to the size of the lump returned by `(funcall seq-elt-fn 0)`.

- **[reader]** **seq-elt-fn** *->seq-barrier (:seq-elt-fn)*

  A function of an `index` argument that returns the lump with that index in some sequence.

- **[accessor]** **seq-indices** *->seq-barrier*

  A sequence of length batch size of indices. The element at index `i` is the index to be passed to `seq-elt-fn` to find the lump whose stripe `i` is copied to stripe `i` of this this lump.

11.5 Utilities

- **[function]** **renormalize-activations** *->v*m-lumps l2-upper-bound*

  If the l2 norm of the incoming weight vector of the a unit is larger than `l2-upper-bound` then renormalize it to `l2-upper-bound`. The list of `->v*m-lumps` is assumed to be eventually fed to the same lump.

  To use it, group the activation clumps into the same GD-OPTIMIZER and hang this function on `after-update-hook`, that latter of which is done for you `arrange-for-renormalizing-activations`.

  See "Improving neural networks by preventing co-adaptation of feature detectors (Hinton, 2012)", http://arxiv.org/pdf/1207.0580.pdf.

- **[function]** **arrange-for-renormalizing-activations** *bpn optimizer l2-upper-bound*

  By pushing a lambda to `after-update-hook` of `optimizer` arrange for all weights beings trained by `optimizer` to be renormalized (as in `renormalize-activations` with `l2-upper-bound`).

  It is assumed that if the weights either belong to an activation lump or are simply added to the activations (i.e. they are biases).

12 Boltzmann Machines

13 Gaussian Processes

14 Natural Language Processing

[in package MGL-NLP]

This in nothing more then a couple of utilities for now which may grow into a more serious toolset for NLP eventually.

- **[function]** **make-n-gram-mappee** *function n*

  Make a function of a single argument that's suitable as the function argument to a mapper function. It calls `function` with every `n` element.

```
(map nil (make-n-gram-mappee #'print 3) '(a b c d e))
..
.. (A B C)
.. (B C D)
.. (C D E)
```

- **[function]** **bleu** *candidates references &key candidate-key reference-key (n 4)*

  Compute the BLEU score for bilingual CORPUS. BLEU measures how good a translation is compared to human reference translations.

  `candidates` (keyed by `candidate-key`) and `references` (keyed by `reference-key`) are sequences of sentences. A sentence is a sequence of words. Words are compared with `equal`, and may be any kind of object (not necessarily strings).

  Currently there is no support for multiple reference translations. `n` determines the largest n-grams to consider.

  The first return value is the `bleu` score (between 0 and 1, not as a percentage). The second value is the sum of the lengths of `candidates` divided by the sum of the lengths of `references` (or `nil`, if the denominator is 0). The third is a list of n-gram precisions (also between 0 and 1 or `nil`), one for each element in [1..`n`].

  This is basically a reimplementation of multi-bleu.perl.

```
(bleu '((1 2 3 4) (a b))
      '((1 2 3 4) (1 2)))
```

```
=> 0.8408964
=> 1
=> (;; 1-gram precision: 4/6
    2/3
    ;; 2-gram precision: 3/4
    3/4
    ;; 3-gram precision: 2/2
    1
    ;; 4-gram precision: 1/1
    1)
```

14.1 Bag of Words

- **[class]** **bag-of-words-encoder**

  `encode` all features of a document with a sparse vector. Get the features of document from `mapper`, encode each feature with `feature-encoder`. `feature-encoder` may return `nil` if the feature is not used. The result is a vector of encoded-feature/value conses. encoded-features are unique (under `encoded-feature-test`) within the vector but are in no particular order.

  Depending on `kind`, value is calculated in various ways:

  - For `:frequency` it is the number of times the corresponding feature was found in `document`.
  - For `:binary` it is always 1.
  - For `:normalized-frequency` and `:normalized-binary` are like the unnormalized counterparts except that as the final step values in the assembled sparse vector are normalized to sum to 1.
  - Finally, `:compacted-binary` is like `:binary` but the return values is not a vector of conses, but a vector of element-type `encoded-feature-type`.

```
(let* ((feature-indexer
         (make-indexer
          (alexandria:alist-hash-table '(("I" . 3) ("me" . 2) ("mine" . 1)))
          2))
       (bag-of-words-encoder
         (make-instance 'bag-of-words-encoder
                        :feature-encoder feature-indexer
                        :feature-mapper (lambda (fn document)
                                          (map nil fn document))
                        :kind :frequency)))
  (encode bag-of-words-encoder '("All" "through" "day" "I" "me" "mine"
                                 "I" "me" "mine" "I" "me" "mine")))
=> #((0 . 3.0d0) (1 . 3.0d0))
```

- **[reader]** **feature-encoder** *bag-of-words-encoder (:feature-encoder)*
- **[reader]** **feature-mapper** *bag-of-words-encoder (:feature-mapper)*

- **[reader]** **encoded-feature-test** *bag-of-words-encoder (:encoded-feature-test = #'eql)*
- **[reader]** **encoded-feature-type** *bag-of-words-encoder (:encoded-feature-type = t)*
- **[reader]** **bag-of-words-kind** *bag-of-words-encoder (:kind = :binary)*

15 Logging

[in package MGL-LOG]

- **[function]** **log-msg** *format &rest args*
  - **[macro]** **with-logging-entry** *(stream) &body body*
  - **[variable]** ***log-file*** *nil*
  - **[variable]** ***log-time*** *t*
  - **[function]** **log-mat-room** *&key (verbose t)*

16 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: DEFTYPEs, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

16.1 Function and Macro Index

16.2 Variable and Constant Index

16.3 Type Index

16.4 Misc Index