
Layered

Release 0.1.4

May 30, 2016

1	layered.activation module	1
2	layered.cost module	3
3	layered.dataset module	5
4	layered.evaluation module	9
5	layered.example module	11
6	layered.gradient module	13
7	layered.network module	15
8	layered.optimization module	17
9	layered.plot module	19
10	layered.problem module	21
11	layered.trainer module	23
12	layered.utility module	25
	Python Module Index	27

layered.activation module

class ActivationBases: `object``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)

Compute the derivative of the cost with respect to the input of this activation function. Outgoing is what this function returned in the forward pass and above is the derivative of the cost with respect to the outgoing activation.

class IdentityBases: `layered.activation.Activation``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)**class Sigmoid**Bases: `layered.activation.Activation``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)**class Relu**Bases: `layered.activation.Activation``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)**class Softmax**Bases: `layered.activation.Activation``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)**class SparseField** (*inhibition=0.05, leaking=0.0*)Bases: `layered.activation.Activation``__call__` (*incoming*)`delta` (*incoming, outgoing, above*)**class SparseRange** (*range_=0.3, function=<layered.activation.Sigmoid object>*)Bases: `layered.activation.Activation`

E%-Max Winner-Take-All.

Binary activation. First, the activation function is applied. Then all neurons within the specified range below the strongest neuron are set to one. All others are set to zero. The gradient is the one of the activation function for active neurons and zero otherwise.

See: A Second Function of Gamma Frequency Oscillations: An E%-Max Winner-Take-All Mechanism Selects Which Cells Fire. (2009)

__call__ (*incoming*)

delta (*incoming, outgoing, above*)

layered.cost module

class Cost

Bases: object

`__call__ (prediction, target)``delta (prediction, target)`**class SquaredError**Bases: `layered.cost.Cost`

Fast and simple cost function.

`__call__ (prediction, target)``delta (prediction, target)`**class CrossEntropy (epsilon=1e-11)**Bases: `layered.cost.Cost`

Logistic cost function used for classification tasks. Learns faster in the beginning than SquaredError because large errors are penalized exponentially. This makes sense in classification since only the best class will be the predicted one.

`__call__ (prediction, target)``delta (prediction, target)`

layered.dataset module

class DatasetBases: `object`**urls** = []**cache** = **True****classmethod** **folder** ()**parse** ()

Subclass responsibility. The filenames of downloaded files will be passed as individual parameters to this function. Therefore, it must accept as many parameters as provided class-site urls. Should return a tuple of training examples and testing examples.

dump ()**load** ()**download** (*url*)**static** **split** (*examples*, *ratio=0.8*)

Utility function that can be used within the `parse()` implementation of sub classes to split a list of example into two lists for training and testing.

class Test (*amount=10*)Bases: `layered.dataset.Dataset`**cache** = **False****parse** ()**download** (*url*)**dump** ()**folder** ()**load** ()**split** (*examples*, *ratio=0.8*)

Utility function that can be used within the `parse()` implementation of sub classes to split a list of example into two lists for training and testing.

urls = []**class Regression** (*amount=10000*, *inputs=10*)Bases: `layered.dataset.Dataset`

Synthetically generated dataset for regression. The task is to predict the sum and product of all the input values. All values are normalized between zero and one.

cache = False

parse ()

download (url)

dump ()

folder ()

load ()

split (examples, ratio=0.8)

Utility function that can be used within the parse() implementation of sub classes to split a list of example into two lists for training and testing.

urls = []

class Modulo (amount=60000, inputs=32, classes=7)

Bases: *layered.dataset.Dataset*

Synthetically generated classification dataset. The task is to predict the modulo classes of random integers encoded as bit arrays of length 32.

cache = False

parse ()

download (url)

dump ()

folder ()

load ()

split (examples, ratio=0.8)

Utility function that can be used within the parse() implementation of sub classes to split a list of example into two lists for training and testing.

urls = []

class Mnist

Bases: *layered.dataset.Dataset*

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. (from <http://yann.lecun.com/exdb/mnist/>)

urls = ['http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz', 'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz']

parse (train_x, train_y, test_x, test_y)

cache = True

download (url)

dump ()

folder ()

load ()

static read (*data, labels*)

split (*examples, ratio=0.8*)

Utility function that can be used within the parse() implementation of sub classes to split a list of example into two lists for training and testing.

layered.evaluation module

compute_costs (*network, weights, cost, examples*)

compute_error (*network, weights, examples*)

layered.example module

class Example (*data, target*)

Bases: object

Immutable class representing one example in a dataset.

data

target

layered.gradient module

class Gradient (*network, cost*)

Bases: object

`__call__` (*weights, example*)

class Backprop (*network, cost*)

Bases: *layered.gradient.Gradient*

Use the backpropagation algorithm to efficiently determine the gradient of the cost function with respect to each individual weight.

`__call__` (*weights, example*)

class NumericalGradient (*network, cost, distance=1e-05*)

Bases: *layered.gradient.Gradient*

Approximate the gradient for each weight individually by sampling the error function slightly above and below the current value of the weight.

`__call__` (*weights, example*)

Modify each weight individually in both directions to calculate a numeric gradient of the weights.

class CheckedBackprop (*network, cost, distance=1e-05, tolerance=1e-08*)

Bases: *layered.gradient.Gradient*

Computes the gradient both analytically through backpropagation and numerically to validate the backpropagation implementation and derivatives of activation functions and cost functions. This is slow by its nature and it's recommended to validate derivatives on small networks.

`__call__` (*weights, example*)

class BatchBackprop (*network, cost*)

Bases: object

Calculate the average gradient over a batch of examples.

`__call__` (*weights, examples*)

class ParallelBackprop (*network, cost, workers=4*)

Bases: object

Alternative to BatchBackprop that yields the same results but utilizes multiprocessing to make use of more than one processor core.

`__call__` (*weights, examples*)

layered.network module

class Layer (*size, activation*)

Bases: object

apply (*incoming*)

Store the incoming activation, apply the activation function and store the result as outgoing activation.

delta (*above*)

The derivative of the activation function at the current state.

class Matrices (*shapes, elements=None*)

Bases: object

__getitem__ (*index*)

__setitem__ (*index, data*)

copy ()

class Network (*layers*)

Bases: object

feed (*weights, data*)

Evaluate the network with alternative weights on the input data and return the output activation.

static forward (*weight, activations*)

static backward (*weight, activations*)

layered.optimization module

class GradientDescent

Bases: object

Adapt the weights in the opposite direction of the gradient to reduce the error.

`__call__(weights, gradient, learning_rate=0.1)`**class Momentum**

Bases: object

Slow down changes of direction in the gradient by aggregating previous values of the gradient and multiplying them in.

`__call__(gradient, rate=0.9)`**class WeightDecay**

Bases: object

Slowly moves each weight closer to zero for regularization. This can help the model to find simpler solutions.

`__call__(weights, rate=0.0001)`**class WeightTying (*groups)**

Bases: object

Constraint groups of slices of the gradient to have the same value by averaging them. Should be applied to the initial weights and each gradient.

`__call__(matrices)`

layered.plot module

```
class Interface (title='', xlabel='', ylabel='', style=None)
```

```
    Bases: object
```

```
        style
```

```
        title
```

```
        xlabel
```

```
        ylabel
```

```
class State
```

```
    Bases: object
```

```
class Window (refresh=0.5)
```

```
    Bases: object
```

```
        register (position, interface)
```

```
        start (work)
```

Hand the main thread to the window and continue work in the provided function. A state is passed as the first argument that contains a *running* flag. The function is expected to exit if the flag becomes false. The flag can also be set to false to stop the window event loop and continue in the main thread after the *start()* call.

```
        stop ()
```

Close the window and stops the worker thread. The main thread will resume with the next command after the *start()* call.

```
        update ()
```

Redraw the figure to show changed data. This is automatically called after *start()* was run.

```
class Plot (title, xlabel, ylabel, style=None, fixed=None)
```

```
    Bases: layered.plot.Interface
```

```
        style
```

```
        title
```

```
        xlabel
```

```
        ylabel
```

```
        __call__ (values)
```

layered.problem module

```
class Problem (content=None)  
    Bases: object  
    parse (definition)
```

layered.trainer module

```
class Trainer (problem, load=None, save=None, visual=False, check=False)  
    Bases: object  
    __call__ ()  
        Train the model and visualize progress.
```

layered.utility module

repeated (*iterable, times*)

batched (*iterable, size*)

averaged (*callable_, batch*)

listify (*fn=None, wrapper=<class 'list'>*)

From <http://stackoverflow.com/a/12377059/1079110>

ensure_folder (*path*)

hstack_lines (*blocks, sep=' '*)

pairwise (*iterable*)

|

layered.activation, 1
layered.cost, 3
layered.dataset, 5
layered.evaluation, 9
layered.example, 11
layered.gradient, 13
layered.network, 15
layered.optimization, 17
layered.plot, 19
layered.problem, 21
layered.trainer, 23
layered.utility, 25

Symbols

__call__() (Activation method), 1
 __call__() (Backprop method), 13
 __call__() (BatchBackprop method), 13
 __call__() (CheckedBackprop method), 13
 __call__() (Cost method), 3
 __call__() (CrossEntropy method), 3
 __call__() (Gradient method), 13
 __call__() (GradientDecent method), 17
 __call__() (Identity method), 1
 __call__() (Momentum method), 17
 __call__() (NumericalGradient method), 13
 __call__() (ParallelBackprop method), 13
 __call__() (Plot method), 19
 __call__() (Relu method), 1
 __call__() (Sigmoid method), 1
 __call__() (Softmax method), 1
 __call__() (SparseField method), 1
 __call__() (SparseRange method), 2
 __call__() (SquaredError method), 3
 __call__() (Trainer method), 23
 __call__() (WeightDecay method), 17
 __call__() (WeightTying method), 17
 __getitem__() (Matrices method), 15
 __setitem__() (Matrices method), 15

A

Activation (class in layered.activation), 1
 apply() (Layer method), 15
 averaged() (in module layered.utility), 25

B

Backprop (class in layered.gradient), 13
 backward() (Network static method), 15
 BatchBackprop (class in layered.gradient), 13
 batched() (in module layered.utility), 25

C

cache (Dataset attribute), 5
 cache (Mnist attribute), 6

cache (Modulo attribute), 6
 cache (Regression attribute), 6
 cache (Test attribute), 5
 CheckedBackprop (class in layered.gradient), 13
 compute_costs() (in module layered.evaluation), 9
 compute_error() (in module layered.evaluation), 9
 copy() (Matrices method), 15
 Cost (class in layered.cost), 3
 CrossEntropy (class in layered.cost), 3

D

data (Example attribute), 11
 Dataset (class in layered.dataset), 5
 delta() (Activation method), 1
 delta() (Cost method), 3
 delta() (CrossEntropy method), 3
 delta() (Identity method), 1
 delta() (Layer method), 15
 delta() (Relu method), 1
 delta() (Sigmoid method), 1
 delta() (Softmax method), 1
 delta() (SparseField method), 1
 delta() (SparseRange method), 2
 delta() (SquaredError method), 3
 download() (Dataset method), 5
 download() (Mnist method), 6
 download() (Modulo method), 6
 download() (Regression method), 6
 download() (Test method), 5
 dump() (Dataset method), 5
 dump() (Mnist method), 6
 dump() (Modulo method), 6
 dump() (Regression method), 6
 dump() (Test method), 5

E

ensure_folder() (in module layered.utility), 25
 Example (class in layered.example), 11

F

feed() (Network method), 15

folder() (layered.dataset.Dataset class method), 5
folder() (Mnist method), 6
folder() (Modulo method), 6
folder() (Regression method), 6
folder() (Test method), 5
forward() (Network static method), 15

G

Gradient (class in layered.gradient), 13
GradientDecent (class in layered.optimization), 17

H

hstack_lines() (in module layered.utility), 25

I

Identity (class in layered.activation), 1
Interface (class in layered.plot), 19

L

Layer (class in layered.network), 15
layered.activation (module), 1
layered.cost (module), 3
layered.dataset (module), 5
layered.evaluation (module), 9
layered.example (module), 11
layered.gradient (module), 13
layered.network (module), 15
layered.optimization (module), 17
layered.plot (module), 19
layered.problem (module), 21
layered.trainer (module), 23
layered.utility (module), 25
listify() (in module layered.utility), 25
load() (Dataset method), 5
load() (Mnist method), 6
load() (Modulo method), 6
load() (Regression method), 6
load() (Test method), 5

M

Matrices (class in layered.network), 15
Mnist (class in layered.dataset), 6
Modulo (class in layered.dataset), 6
Momentum (class in layered.optimization), 17

N

Network (class in layered.network), 15
NumericalGradient (class in layered.gradient), 13

P

pairwise() (in module layered.utility), 25
ParallelBackprop (class in layered.gradient), 13
parse() (Dataset method), 5

parse() (Mnist method), 6
parse() (Modulo method), 6
parse() (Problem method), 21
parse() (Regression method), 6
parse() (Test method), 5
Plot (class in layered.plot), 19
Problem (class in layered.problem), 21

R

read() (Mnist static method), 6
register() (Window method), 19
Regression (class in layered.dataset), 5
Relu (class in layered.activation), 1
repeated() (in module layered.utility), 25

S

Sigmoid (class in layered.activation), 1
Softmax (class in layered.activation), 1
SparseField (class in layered.activation), 1
SparseRange (class in layered.activation), 1
split() (Dataset static method), 5
split() (Mnist method), 7
split() (Modulo method), 6
split() (Regression method), 6
split() (Test method), 5
SquaredError (class in layered.cost), 3
start() (Window method), 19
State (class in layered.plot), 19
stop() (Window method), 19
style (Interface attribute), 19
style (Plot attribute), 19

T

target (Example attribute), 11
Test (class in layered.dataset), 5
title (Interface attribute), 19
title (Plot attribute), 19
Trainer (class in layered.trainer), 23

U

update() (Window method), 19
urls (Dataset attribute), 5
urls (Mnist attribute), 6
urls (Modulo attribute), 6
urls (Regression attribute), 6
urls (Test attribute), 5

W

WeightDecay (class in layered.optimization), 17
WeightTying (class in layered.optimization), 17
Window (class in layered.plot), 19

X

xlabel (Interface attribute), 19

xlabel (Plot attribute), 19

Y

ylabel (Interface attribute), 19

ylabel (Plot attribute), 19