Layered

Release 0.1.4

May 30, 2016

Contents

CHAPTER 1

layered.activation module

class Activation
Bases: 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 Identity
Bases: *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*)

CHAPTER 2

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*)

CHAPTER 3

layered.dataset module

class `Dataset`

Bases: `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*

Sythetically 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-id

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.

CHAPTER 4

layered.evaluation module

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

compute_error (*network*, *weights*, *examples*)

CHAPTER 5

layered.example module

class **Example**(*data*, *target*)

Bases: `object`

Immutable class representing one example in a dataset.

data

target

CHAPTER 6

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 trough 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*)

CHAPTER 7

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*)

CHAPTER 8

layered.optimization module

class GradientDecent
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*)

CHAPTER 9

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*)

CHAPTER 10

layered.problem module

class **Problem**(*content=None*)

Bases: object

parse(*definition*)

CHAPTER 11

layered.trainer module

class **Trainer**(*problem*, *load=None*, *save=None*, *visual=False*, *check=False*)

Bases: object

__call__()

Train the model and visualize progress.

CHAPTER 12

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*)

l

Symbols

A

B

C

D

E

F

Y