MyGrad Documentation

Release 2.2.0

Ryan Soklaski

Feb 18, 2023

CONTENTS:

1 "Drop in" automatic differentiation? 3

2 MyGrad aims for parity with NumPy's major features 5

3 What About JAX? 7

3.1 Installing MyGrad 7
3.2 Introducing MyGrad 8
3.3 MyGrad's Tensor 11
3.4 Views and In-Place Operations 25
3.5 Performance Tips 26
3.6 Writing Your Own Operations 30
3.7 Tensor creation routines (`mygrad.tensor_creation`) 34
3.8 Tensor manipulation routines (`mygrad.tensor_manip`) 55
3.9 Linear algebra (`mygrad.linalg`) 71
3.10 Mathematical functions (`mygrad.math`) 82
3.11 Indexing Routines (`mygrad.indexing_routines`) 181
3.12 Neural network operations (`mygrad.nnet`) 183
3.13 Input and Output 224
3.14 Computational graph visualization(`mygrad.computational_graph`) 226
3.15 Changelog 227

Bibliography 241

Index 245

MyGrad is a lightweight library that adds automatic differentiation to NumPy – its only dependency is NumPy. Simply "drop in" a MyGrad tensor into your NumPy-based code, and start differentiating!

```
>>> import mygrad as mg
>>> import numpy as np

>>> x = mg.tensor([1., 2., 3.])  # like numpy.array, but supports backprop
>>> f = np.sum(x * x)  # tensors can be passed directly to native numpy functions!
>>> f.backward() # triggers automatic differentiation
>>> x.grad  # stores [df/dx0, df/dx1, df/dx2]
array([2., 4., 6.])
```

MyGrad's primary goal is to make automatic differentiation accessible and easy to use across the Python/NumPy ecosystem. As such, it strives to behave and feel exactly like NumPy so that users need not learn yet another array-based math library.

Of the various modes and flavors of auto-diff, MyGrad currently only supports back-propagation from a scalar quantity.

CHAPTER

ONE

"DROP IN" AUTOMATIC DIFFERENTIATION?

What we mean by drop in automatic differentiation is that you can take a third party function, which is written in NumPy, and pass MyGrad tensors as its inputs – this will coerce it into using MyGrad functions internally so that we can differentiate the function.

Listing 1: What we mean by drop in autodiff

```python
from third_party_lib import some_numpy_func

import mygrad as mg

arr1 = mg.tensor(...) # some MyGrad Tensor (instead of a NumPy array)
arr2 = mg.tensor(...) # some MyGrad Tensor (instead of a NumPy array)

output = some_numpy_func(arr1, arr2)  # "drop in" the MyGrad tensors

output.backward()  # output is a MyGrad tensor, not a NumPy array!

arr1.grad  # stores d(some_numpy_func) / d(arr1)
arr2.grad  # stores d(some_numpy_func) / d(arr2)
```

CHAPTER

TWO

MYGRAD AIMS FOR PARITY WITH NUMPY'S MAJOR FEATURES

NumPy's ufuncs are richly supported. We can even differentiate through an operation that occur in-place on a tensor and applies a boolean mask to the results:

```
>>> x = mg.tensor([1., 2., 3.])
>>> y = mg.zeros_like(x)
>>> np.multiply(x, x, where=[True, False, True], out=y)
>>> y.backward()
>>> x.grad
array([2., 0., 6.])
```

NumPy's view semantics are also mirrored to a high fidelity: performing basic indexing and similar operations on tensors will produce a "view" of that tensor's data, thus a tensor and its view share memory. This relationship will also manifest between the derivatives stored by a tensor and its views!

```
>>> x = mg.arange(9.).reshape(3, 3)
>>> diag_view = np.einsum("ii->i", x)  # returns a view of the diagonal elements of `x`
>>> x, diag_view
(Tensor([[0., 1., 2.],
[3., 4., 5.],
[6., 7., 8.]]),
Tensor([0., 4., 8.]))

# views share memory
>>> np.shares_memory(x, diag_view)
True

# mutating a view affects its base (and all other views)
>>> diag_view *= -1  # mutates x in-place
>>> x
Tensor([[-0.,  1.,  2.],
        [ 3., -4.,  5.],
        [ 6.,  7., -8.]])

>>> (x ** 2).backward()
>>> x.grad, diag_view.grad
(array([[ -0.,   2.,   4.],
        [  6.,  -8.,  10.],
        [ 12.,  14., -16.]]),
 array([ -0.,  -8., -16.]))

# the gradients have the same view relationship!
```

(continues on next page)

(continued from previous page)

```
>>> np.shares_memory(x.grad, diag_view.grad)
True
```

Basic and advanced indexing is fully supported

```
>>> (x[x < 4] ** 2).backward()
>>> x.grad
array([[0., 2., 4.],
       [6., 0., 0.],
       [0., 0., 0.]])
```

NumPy arrays and other array-likes play nicely with MyGrad's tensor. These behave like constants during automatic differentiation

```
>>> x = mg.tensor([1., 2., 3.])
>>> constant = [-1., 0., 10]  # can be a numpy array, list, or any other array-like
>>> (x * constant).backward()  # all array-likes are treated as constants
>>> x.grad
array([-1.,  0., 10.])
```

CHAPTER

THREE

WHAT ABOUT JAX?

Doesn't JAX already provide drop in automatic differentiation? Not quite; JAX provides *swap-out* automatic differentiation: you must swap out the version of NumPy you are using *before* you write your code. Thus you cannot simply differentiate some third party function by passing it a JAX array.

"Is MyGrad a competitor to JAX? Should I stop using JAX and start using MyGrad?"

Goodness gracious, no! MyGrad is *not* meant to compete with the likes of JAX, which offers far more functionality in the way of computing higher-order derivatives, Jacobian vector projects, in terms of providing a jit… this list goes on. MyGrad is meant to be a simple and highly accessible way to provide basic automatic differentiation capabilities to the NumPy ecosystem. Anyone who knows how to use NumPy can very easily learn to use MyGrad. It is especially great for teaching. But once your auto-diff needs extend beyond derivatives of scalars, it is time to graduate to JAX.

3.1 Installing MyGrad

MyGrad requires numpy. It is highly recommended that you utilize numpy built with MKL for access to optimized math routines (e.g. install numpy via anaconda). You can install MyGrad using pip:

```
pip install mygrad
```

You can instead install MyGrad from its source code. Clone this repository and navigate to the MyGrad directory, then run:

```
pip install .
```

3.1.1 Support for Python and NumPy

MyGrad abides by the NEP 29 recommendation, and adopts a common "time window-based" policy for support of NumPy versions. Accordingly, MyGrad's drop schedule for NumPy versions can be found here.

Note, however, that MyGrad will maintain a wider window of support for minor Python versions than is specified by NEP 29. Because our only dependency is NumPy, and because we strive to remain an exceptionally lightweight and flexible dependency to our users, we will support minor versions of Python until their end of life, *or* until our lowest supported version of NumPy drops support for that version of Python – whichever occurs first.

3.2 Introducing MyGrad

MyGrad is a lightweight library that adds automatic differentiation to NumPy – its only dependency is NumPy!

```
>>> import mygrad as mg
>>> import numpy as np

>>> x = mg.tensor([1., 2., 3.])  # like numpy.array, but supports backprop!
>>> f = np.sum(x * x)  # tensors work with numpy functions!
>>> f.backward() # triggers automatic differentiation
>>> x.grad  # stores [df/dx0, df/dx1, df/dx2]
array([2., 4., 6.])
```

Its primary goal is to make automatic differentiation an accessible and easy to use across the Python/NumPy ecosystem. As such, it strives to behave and feel exactly like NumPy so that users need not learn yet another array-based math library. You can pass MyGrad's `Tensor` to NumPy's functions in order to make them differentiable! Of the various modes and flavors of auto-diff, MyGrad supports backpropagation from a scalar quantity.

3.2.1 A Simple Application

Let's use `mygrad` to compute the derivative of $f(x) = x^2$ evaluated at $x = 3$ (which is $\frac{df}{dx}\rvert_{x=3} = 2 \times 3$).

`Tensor` behaves nearly identically to NumPy's ndarray, in addition to having the machinery needed to compute the analytic derivatives of functions. Suppose we want to compute this derivative at `x = 3`. We can create a 0-dimensional tensor (a scalar) for x and compute `f(x)`:

```
>>> import mygrad as mg
>>> import numpy as np
>>> x = mg.tensor(3.0)
>>> f = np.square(x)  # mygrad's tensors can be passed into NumPy functions
>>> f
Tensor(9.0)
```

Invoking *`backward()`* on `f` instructs `mygrad` to trace through the computational graph that produced `f` and compute the derivatives of `f` with respect to all of its independent variables. Thus, executing `f.backward()` will compute $\frac{df}{dx} = 2x$ at $x = 3$, and will store the resulting value in `x.grad`:

```
>>> f.backward()  # triggers computation of ``df/dx``
>>> x.grad  # df/dx = 2x = 6.0
array(6.0)
```

This is the absolute tip of the iceberg. `mygrad` can compute derivatives of multivariable composite functions of tensor-valued variables!

3.2.2 Gradient Descent with MyGrad

Performing gradient descent on $L(w) = w^2$

```
w = mg.tensor(10.0)
learning_rate = 0.3
num_steps = 10
print(w)

for step_cnt in range(num_steps):
      = w ** 2    # compute L(w) (this also "nulls" any derivatives")
    .backward()  # compute derivative of L

    # Update w via gradient-step..
    # We do an augmented update on the underlying numpy-array
    # stored by `w`
    w.data -= learning_rate * w.grad
    print(w)
```

The following steps are printed out.. see that gradient descent leads us towards the minimum of $w = 0$

```
Tensor(10.)
Tensor(4.)
Tensor(1.6)
Tensor(0.64)
Tensor(0.256)
Tensor(0.1024)
Tensor(0.04096)
Tensor(0.016384)
Tensor(0.0065536)
Tensor(0.00262144)
Tensor(0.00104858)
```

3.2.3 Some Bells and Whistles

`mygrad` supports all of NumPy's essential features, including:

- N-dimensional tensors that can be reshaped and have their axes transposed
- creating and operating on views of tensors
- in-place operations on tensors
- vectorization
- broadcasting
- basic and advanced indexing (including all varieties of mixed indexing schemes) for both getting and setting items.
- fully-fledged support for einsum (including broadcasting and traces)

`Tensor` plays nicely with NumPy-arrays, which behave as constants when they are used in computational graphs:

```
>>> import numpy as np
>>> x = mg.tensor([2.0, 2.0, 2.0])
>>> y = np.array([1.0, 2.0, 3.0])
>>> f = x ** y  # (2 ** 1, 2 ** 2, 2 ** 3)
>>> f.backward()
>>> x.grad
array([ 1.,  4., 12.])
```

`nnet` supplies essential functions for machine learning, including:

- N-dimensional convolutions (with striding, dilation, and padding)
- N-dimensional pooling
- A gated recurrent unit for sequence-learning (with input-level dropout and variational hidden-hidden dropout)

It leverages a nice sliding window view function, which produces convolution-style windowed views of arrays/tensors without making copies of them, to intuitively (and quite efficiently) perform the neural network-style convolutions and pooling.

3.2.4 Advanced Example

The following is an example of using `mygrad` to compute the hinge loss of classification scores and to "back-propagate" through (compute the gradient of) this loss. This example demonstrates some of mygrad's ability to perform back-propagation through broadcasted operations, basic indexing, advanced indexing, and in-place assignments.

```
>>> from mygrad import Tensor
>>> import numpy as np
>>> class_scores = Tensor(10 * np.random.rand(100, 10))         # 100 samples, 10␣
↪possible classes for each
>>> class_labels = np.random.randint(low=0, high=10, size=100)  # correct label for each␣
↪datum
>>> class_labels = (range(len(class_labels)), class_labels)
>>> correct_class_scores = class_scores[class_labels]

>>> Lij = class_scores - correct_class_scores[:, np.newaxis] + 1. 0 # 100x10 margins
>>> Lij[Lij <= 0] = 0      # scores within the hinge incur no loss
>>> Lij[class_labels] = 0  # the score corresponding to the correct label incurs no loss

>>> loss = Lij.sum() / class_scores.shape[0]  # compute mean hinge loss
>>> loss.backward()    # compute gradient of loss w.r.t all dependent tensors
>>> class_scores.grad  # d(loss)/d(class_scores)
array([[ 0.  ,  0.01,  0.  , -0.04,  0.  ,  0.  ,  0.01,  0.  ,  0.01, 0.01], ...])
```

3.2.5 Computational Graph Visualization

MyGrad provides the capability to visually render diagrams of your computational graphs:

```
import mygrad as mg
from mygrad.computational_graph import build_graph
x = mg.tensor(2)
y = mg.tensor(3)
f = x * y
g = f + x - 2

build_graph(g, names=locals())
```

mygrad uses Graphviz and a Python interface for Graphviz to render the computational graphs built using tensors. These graphs can be rendered in Jupyter notebooks, allowing for quick checks of graph structure, or can be saved to file for later reference.

The dependencies can be installed with:

```
conda install graphviz
conda install python-graphviz
```

Big thanks to Petar Griggs for implementing these fantastic viz capabilities!

3.3 MyGrad's Tensor

`Tensor` is the most critical piece of MyGrad. It is a numpy-array-like object capable of serving as a node in a computational graph that supports back-propagation of derivatives via the chain rule.

You can effectively do a drop-in replacement of a numpy array with a `Tensor` for all basic mathematical operations. This includes basic and advanced indexing, broadcasting, sums over axes, etc; it will simply just work.

```
>>> import mygrad as mg  # note that we replace numpy with mygrad here
>>> x = mg.arange(9).reshape(3, 3)
>>> x
Tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
>>> y = x[x == 4] ** 2
>>> y
Tensor([16], dtype=int32)
```

Thus MyGrad users can spend their time mastering numpy and their skills will transfer seamlessly when using this autograd library.

3.3.1 Creating a Tensor

`Tensor` can be passed any "array-like" object of numerical data. This includes numbers, sequences (e.g. lists), nested sequences, numpy-ndarrays, and other mygrad-tensors. mygrad also provides familiar numpy-style tensor-creation functions (e.g. *arange()*, *linspace()*, etc.)

```
>>> import mygrad as mg
>>> mg.tensor(2.3)  # creating a 0-dimensional tensor
Tensor(2.3)
>>> mg.tensor(np.array([1.2, 3.0]))  # casting a numpy-array to a tensor
Tensor([1.2, 3.0])
>>> mg.tensor([[1, 2], [3, 4]])  # creating a 2-dimensional tensor from lists
Tensor([[1, 2],
       [3, 4]])
>>> mg.arange(4)    # using numpy-style tensor creation functions
Tensor([0, 1, 2, 3])
```

Integer-valued tensors are treated as constants

```
>>> mg.astensor(1, dtype=np.int8).constant
True
```

By default, float-valued tensors are not treated as constants

```
>>> mg.astensor(1, dtype=np.float32).constant
False
```

3.3.2 Forward and Back-Propagation

Let's construct a computational graph consisting of two zero-dimensional tensors, `x` and `y`, which are used to compute an output tensor, . This is a "forward pass imperative" style for creating a computational graph - the graph is constructed as we carry out the forward-pass computation.

```
>>> x = Tensor(3.0)
>>> y = Tensor(2.0)
>>>  = 2 * x + y ** 2
```

Invoking `.backward()` signals the computational graph to compute the total-derivative of with respect to each one of its dependent variables. I.e. `x.grad` will store `d/dx` and `y.grad` will store `d/dy`. Thus we have back-propagated a gradient from through our graph.

Each tensor of derivatives is computed elementwise. That is, if `x = Tensor(x0, x1, x2)`, then `d/dx` represents `[d/d(x0), d/d(x1), d/d(x2)]`

```
>>> .backward()  # computes d/dx and d/dy
>>> x.grad  # d/dx
array(6.0)
>>> y.grad  # d/dy
array(4.0)
>>> .grad
array(1.0)  # d/d
```

Once the gradients are computed, the computational graph containing `x`, `y`, and is cleared automatically. Additionally, involving any of these tensors in a new computational graph will automatically null their gradients.

```
>>> 2 * x
>>> x.grad is None
True
```

Or, you can use the *null_grad()* method to manually clear a tensor's gradient

```
>>> y.null_grad()
Tensor(2.)
>>> y.grad is None
True
```

3.3.3 Accessing the Underlying NumPy Array

`Tensor` is a thin wrapper on `numpy.ndarray`. A tensor's underlying numpy-array can be accessed via `.data`. This returns a direct reference to the numpy array.

```
>>> x = mg.tensor([1, 2])
>>> x.data
array([1, 2])
```

```
>>> import numpy as np
>>> np.asarray(x)
array([1, 2])
```

3.3.4 Producing a "View" of a Tensor

MyGrad's tensors exhibit the same view semantics and memory-sharing relationships as NumPy arrays. I.e. any (non-scalar) tensor produced via basic indexing will share memory with its parent.

```
>>> x = mg.tensor([1., 2., 3., 4.])
>>> y = x[:2]  # the view: Tensor([1., 2.])
>>> y.base is x
True
>>> np.shares_memory(x, y)
True
```

Mutating shared data will propagate through views:

```
>>> y *= -1
>>> x
Tensor([-1., -2.,  3.,  4.])
>>> y
Tensor([-1., -2.])
```

And this view relationship will also manifest between the tensors' gradients

```
>>> (x ** 2).backward()
>>> x.grad
array([-2., -4.,  6.,  8.])
>>> y.grad
array([-2., -4.])
```

3.3.5 Documentation for mygrad.Tensor

Tensor.astype(dtype[, casting, copy, constant])	Copy of the tensor with the specified dtype.
Tensor.backward([grad])	Trigger backpropagation and compute the derivatives of this tensor.
Tensor.base	A reference to the base tensor that the present tensor is a view of.
Tensor.clear_graph()	Removes the current tensor – and tensors above it – from their shared computational graph.
Tensor.constant	If `True`, this tensor is a constant; it will not propagate any gradient.
Tensor.copy(*[, constant])	Produces a copy of `self` with `copy.creator=None`.
Tensor.creator	The `Operation` instance that produced `self`.
Tensor.dtype	Data-type of the tensor's elements.
Tensor.grad	Returns the derivative of with respect to this tensor.
Tensor.item()	Copy an element of a tensor to a standard Python scalar and return it.
Tensor.ndim	Number of tensor dimensions.
Tensor.null_grad(*[, _clear_view_info])	Sets this tensor's gradient to be `None`.
Tensor.null_gradients([clear_graph])	**Deprecated: Tensors will automatically have their computational graphs cleared during backprop.
Tensor.shape	Tuple of tensor dimension-sizes.
Tensor.size	Number of elements in the tensor.
Tensor.T	Same as self.transpose(), except that self is returned if self.ndim < 2 and a view of the underlying data is utilized whenever possible.

mygrad.Tensor.astype

`Tensor.astype`(*dtype: DTypeLikeReals*, *casting='unsafe'*, *copy: bool = True*, ***, *constant: Optional[bool] = None*) → Tensor

Copy of the tensor with the specified dtype.

The resulting tensor is not involved in any computational graph and has no gradient associated with it.

This docstring was adapted from that of `ndarray.astype`.

Parameters

dtype
[Union[type, str]] The real-valued numeric data type. This can be a numpy dtype or a corresponding string identifier.

casting
[Literal['no', 'equiv', 'safe', 'same_kind', 'unsafe']]

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.

- ‘unsafe’ means any data conversions may be done.

copy
[bool, optional (default=True)] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype` and `constant` requirements are satisfied, the input tensor is returned instead of a copy.

constant
[Optional[bool]] If specified, determines if the returned tensor is a constant. Otherwise this argument is inferred from the original tensor.

Returns

Tensor
The resulting tensor with the specified data type.

References

[1].. Retrieved from: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.astype.html

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> x = mg.arange(10); x
Tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Using a string to specify the data type:

```
>>> x.astype("float32")
Tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], dtype=float32)
```

Specifying a numpy data type object, and specifying that the tensor is to be treated as a constant:

```
>>> x.astype(np.int8, constant=True)
Tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int8)
```

mygrad.Tensor.backward

`Tensor.`**`backward`**(*grad: Optional[ArrayLike] = None*)

Trigger backpropagation and compute the derivatives of this tensor.

Designating this tensor as the tensor , compute d/dx for all (non-constant) tensors that preceded in its computational graph, and store each of these derivatives in `x.grad` respectively.

Once back-propagation is finished, the present tensor is removed from all computational graphs, and the preceding graph is cleared.

If is a non-scalar tensor (i.e. `.ndim` is greater than 0), then calling `.backward()` will behave as if was first reduced to a scalar via summation. I.e. it will behave identically to `.sum().backward()`; this ensures that each element of any d/dx will represent a derivative of a scalar function.

Parameters

grad
[Optional[array_like], (must be broadcast-compatible with `self`] By default, the present tensor is treated as the terminus of the computational graph (). Otherwise, one can specify a "downstream" derivative, representing `d/d(self)`. This can be used to effectively connect otherwise separate computational graphs.

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor(2)
>>> y = mg.tensor(3)
>>> w = x * y
>>>  = 2 * w
>>> .backward()  # computes d/d, d/dw, d/dy, and d/dx
```

```
>>> .grad  # d/df == 1 by identity
array(1.)
>>> w.grad  # d/dw
array(2.)
>>> y.grad # d/dy = d/dw * dw/dy
array(4.)
>>> x.grad # d/dx = d/dw * dw/dx
array(6.)
```

Calling `.backward()` from a non-scalar tensor is equivalent to first summing that tensor.

```
>>> tensor = mg.tensor([2.0, 4.0, 8.0])
>>>  = tensor * tensor[::-1]  # [x0*x2, x1*x1, x2*x0]
>>> .backward()  # behaves like  = x0*x2 + x1*x1 + x2*x0
>>> tensor.grad
array([16.,  8.,  4.])
```

```
>>> tensor = mg.Tensor([2.0, 4.0, 8.0])
>>>  = tensor * tensor[::-1]
>>> .sum().backward()
>>> tensor.grad
array([16.,  8.,  4.])
```

Specifying a value for `grad`

```
>>> x = mg.Tensor(1.)
>>> x.backward(2.)
>>> x.grad  # Would normally be d/d == 1
array(2.)
```

mygrad.Tensor.base

property Tensor.**base**: Optional[Tensor]

A reference to the base tensor that the present tensor is a view of.

It this tensor owns its memory, then this returns `None`.

Examples

The base of a tensor that owns its memory is `None`:

```
>>> import mygrad as mg
>>> x = mg.arange(5)
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
>>> y.data.base is x.data
True
```

A view of a view has the same base as its "parent"

```
>>> z = y[:]
>>> z.base is x
True
```

The behavior of `Tensor.base` departs from that of `ndarray.base` in that mygrad will never create an "internal" tensor to serve as a base; e.g.

```
>>> import numpy as np
>>> np.reshape(2., (1,)).base
array(2.)
```

```
>>> mg.reshape(2., (1,)).base is None
True
```

mygrad.Tensor.clear_graph

Tensor.**clear_graph**()

Removes the current tensor – and tensors above it – from their shared computational graph.

This de-references all operations involved in the graph and the intermediate tensors that were created by it. Arrays whose memory were locked by the computational graph will have their writeability restored.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> x = np.array([1., 2.])
>>> y = mg.multiply(2., x)
>>> x.flags.writeable, y.creator
(False, <mygrad.math.arithmetic.ops.Multiply at 0x224f89cac48>)
>>> y.clear_graph()
>>> x.flags.writeable, y.creator
(True, None)
```

mygrad.Tensor.constant

property Tensor.**constant:** bool

If True, this tensor is a constant; it will not propagate any gradient.

Additionally, any tensor that is a descendant of constant tensors will also be a constant.

Integer-valued tesnors, Python scalars and NumPy arrays are treated as constant tensors when included in My-Grad computational graphs.

Returns

bool

Examples

Constant-tensors do not back-propagate gradients:

```
>>> import mygrad as mg
>>> x = mg.Tensor([1., 2.], constant=True)
>>> y = mg.Tensor([0., 3.], constant=False)
>>> f = x * y
>>> f.backward()
```

```
>>> x.grad is None  # x has no gradient
True
>>> y.grad
array([1., 2.])
```

A tensor that is derived solely from constant tensors is also a constant:

```
>>> import numpy as np
>>> x = mg.Tensor([1., 2.], constant=True)
>>> y = mg.Tensor([0., 3.], constant=True)
>>> z = (x + y) ** 2 - np.array([8., 7.])
>>> z.constant
True
```

Integer-valued tensors are treated as constants

```
>>> mg.Tensor([1, 2]).constant
True
```

mygrad.Tensor.copy

`Tensor.copy(*, constant: Optional[bool] = None)` → Tensor

Produces a copy of `self` with `copy.creator=None`.

Copies of the underlying numpy data array and gradient array are created.

No information regarding the tensor's participation in the computational graph are copied.

Parameters

constant
[Optional[bool]]

Returns

Tensor

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor(data, constant=constant)
>>> y = x * 2
>>> y.backward()
>>> y_copy = y.copy()
>>> y_copy
Tensor(6)
>>> y_copy.grad
array(1.)
>>> y_copy.creator is None
True
```

mygrad.Tensor.creator

`property Tensor.creator: Optional[Operation]`

The `Operation` instance that produced `self`.

Returns

creator
[Optional[Operation]] The operation-instance that created the tensor, or *None*.

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor(3)
>>> x.creator is None
True
>>> y = mg.Tensor(2)
>>> z = x * y  # Multiply(x, y) -> z
>>> z.creator
 <mygrad.math.arithmetic.ops.Multiply at 0x2df5a130438>
```

mygrad.Tensor.dtype

property Tensor.**dtype: dtype**

Data-type of the tensor's elements.

Returns

numpy dtype object

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([[0, 1],
...                [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

mygrad.Tensor.grad

property Tensor.**grad: Optional[ndarray]**

Returns the derivative of with respect to this tensor.

 is the terminal node in the compuational graph from which `.backward()` was invoked.

If this tensor is a view of another tensor then their gradients will exhibit the same memory-sharing relationship as their data.

Returns

d/dx: numpy.ndarray
The gradient of the terminal node in a computational graph with respect to this tensor. The shape of this numpy array matches `self.shape`

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([1.0, 2.0])
```

Prior to backpropagation tensors have `None` set for their gradients.

```
>>> x.grad is None
True
```

Now we trigger backpropagation…

```
>>>   = x ** 2
>>> .backward()
```

and we see that `x.grad` stores d/dx

```
>>> x.grad  # d/dx
array([2., 4.])
```

Now we will demonstrate the relationship between gradient a view tensor and that of its base.

```
>>> base = mg.Tensor([1.0, 2.0, 3.0])
>>> view = base[:2]; view
Tensor([1., 2.])
```

```
>>>  = base ** 2
>>> .backward()
```

Although `view` is not directly involved in the computation in , and thus would not typically store a gradient in due to `.backward()`, it shares memory with `base` and thus it stores a gradient in correspondence to this "view relationship". I.e. because `view == base[:2]`, then we expect to find that `view.grad == base.grad[:2]`.

```
>>> base.grad
array([2., 4., 6.])
>>> view.grad
array([2., 4.])
```

```
>>> view.grad.base is base.grad
True
```

The reasoning here is that, because a base tensor and its view share the same array data, then varying an element in that data implies that both the base tensor and the view will change (assuming the variation occurs specifically in a shared region). It follows that the base tensor's gradient must share the same relationship with the view-tensor since these are measures of "cause and effects" associated with varying elements of data (albeit infinitesmaly).

mygrad.Tensor.item

`Tensor.item()` → Union[int, float]

Copy an element of a tensor to a standard Python scalar and return it.

Note that the returned object does not support back-propagation.

Returns

z

[Standard Python scalar object] A copy of the specified element of the tensor as a suitable Python scalar

Examples

```
>>> import mygrad as mg
>>> x = Tensor([22.2])
>>> x.item()
22.2
>>> type(x.item())
float
```

mygrad.Tensor.ndim

property Tensor.**ndim**: int

Number of tensor dimensions. I.e. the number of indices that must be supplied to uniquely specify an element in the tensor.

Returns

int

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([1, 2, 3])
>>> x.ndim
1
>>> x[0]  # a single index identifies an element in `x`
Tensor(1)
```

```
>>> y = mg.Tensor([[1, 2, 3],
...                [4, 5, 6]])
>>> y.ndim
2
>>> y[0, 0]  # two indices are required to identify an element in `x`
Tensor(1)
```

mygrad.Tensor.null_grad

Tensor.**null_grad**(*, *_clear_view_info: bool = False*) → Tensor

Sets this tensor's gradient to be `None`.

This operation is performed in-place, but a reference to the tensor is returned in order to permit mapping semantics.

Also removes any `base` reference from disconnected views.

Returns

self

Examples

```
>>> import  mygrad as mg
>>> x = mg.Tensor(2.)
>>> (x ** 2).backward()
>>> x.grad
array(4.)
>>> x.null_grad()  # returns a reference of `x`
Tensor(2.0)
>>> x.grad is None
True
```

mygrad.Tensor.null_gradients

`Tensor.null_gradients(clear_graph: bool = True)`

Deprecated: Tensors will automatically have their computational graphs cleared during backprop. Simply involving a tensor in a new computational graph will null its gradient.

Sets the gradient for this tensor and for all preceding tensors in the computation graph to `None`.

Additionally, the computational graph that terminates in this tensor can also be cleared during this process.

Parameters

clear_graph
[bool, optional (default=True)] If `True` clear the computational graph in addition to nulling the gradients.

Notes

It is advised to clear the computational graph when nulling gradients, i.e. invoke `null_gradients(clear_graph=True)` (or simply `null_gradients()`). This de-references all intermediate operations and tensors in the computational graph and thus permits garbage collection - freeing the memory that was used by the computational graph.

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor(2)
>>> y = mg.tensor(3)
>>> w = x * y
>>> f = 2 * w
>>> f.backward()  # computes df/df, df/dw, df/dy, and df/dx
>>> any(tensor.grad is None for tensor in (f, w , x, y))
False
```

```
>>> f.null_gradients()  # set tensor.grad to None for all tensors in the graph
>>> all(tensor.grad is None for tensor in (f, w , x, y))
True
```

mygrad.Tensor.shape

`property Tensor.shape: Shape`

Tuple of tensor dimension-sizes.

Sizes are reported in row-major order.

Returns

Tuple[int, ...]

See also:

`mygrad.reshape`
similar function

Tensor.reshape
similar method

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([1, 2, 3, 4])  # axis-0 has size 4
>>> x.shape
(4,)
>>> y = mg.Tensor([[1, 2, 3],    # axis-0 has size 2, axis-1 has size 3
...                [4, 5, 6]])
>>> y.shape
(2, 3)
```

The shape attribute can also be set to reshape the tensor in-place

```
>>> y.shape = (1, 6, 1)
>>> y
Tensor([[[1],
         [2],
         [3],
         [4],
         [5],
         [6]]])
```

mygrad.Tensor.size

property Tensor.**size**: int

Number of elements in the tensor. i.e., the product of the tensor's dimensions.

Returns

int

Examples

```
>>> import mygrad as mg
>>> x = mg.zeros((3, 5, 2))  # creates a tensor with 3x5x2 (= 30) elements
>>> x.size
30
```

mygrad.Tensor.T

property Tensor.**T**: **Tensor**

Same as self.transpose(), except that self is returned if self.ndim < 2 and a view of the underlying data is utilized whenever possible.

Returns

Tensor

Examples

```
>>> import mygrad as mg
>>> y = mg.Tensor([[1, 2, 3],
...                [4, 5, 6]])
>>> y.T
Tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

3.4 Views and In-Place Operations

3.4.1 Producing a "View" of a Tensor

MyGrad's tensors exhibit the same view semantics and memory-sharing relationships as NumPy arrays. I.e. any (non-scalar) tensor produced via basic indexing will share memory with its parent.

```
>>> x = mg.tensor([1., 2., 3., 4.])
>>> y = x[:2]  # the view: Tensor([1., 2.])
>>> y.base is x
True
>>> np.shares_memory(x, y)
True
```

Mutating shared data will propagate through views:

```
>>> y *= -1
>>> x
Tensor([-1., -2.,  3.,  4.])
>>> y
Tensor([-1., -2.])
```

And this view relationship will also manifest between the tensors' gradients

```
>>> (x ** 2).backward()
>>> x.grad
array([-2., -4.,  6.,  8.])
>>> y.grad
array([-2., -4.])
```

3.4.2 In-Place Operations are not Efficient

It is important to note that although MyGrad's view semantics promote a rich parity with NumPy, certain aspects should be avoided in the interest of optimized performance. Namely, performing in-place operations on tensors is generally not more efficient than their non-mutating counterparts.

This is because MyGrad has to track the state of tensors that are involved in a computational graph. Thus a mutated tensor must have its pre-augmented state stored for future reference; this defeats the performance benefit of writing to an array's memory in-place. This is especially inefficient if you are mutating a tensor involved with multiple views of the same memory(By contrast, producing a view of a tensor *is* efficient as one would expect).

Thus these NumPy-like in-place semantics are supported by MyGrad not for the same performance purposes, but instead to support convenient and familiar code-patterns and to enable one to port NumPy code to MyGrad (or, in the future, inject MyGrad tensors into NumPy!!) and get the exact same behavior.

A final note: MyGrad's in-place operations, when run under *no_autodiff()* mode, do not incur the extra costs noted above, and thus your code will benefit from the performance benefits of in-place operations.

3.5 Performance Tips

The following functions provide users with controls for optimizing MyGrad code by either suspending its memory-guarding behavior or by disabling automatic differentiation altogether. These are important utilities for speeding up your code.

Beyond the points made below, general performance tips for NumPy – e.g. leveraging vectorized operations, heeding NumPy's row-major memory layout for arrays when constructing tensors, and using basic indexing to create views of arrays instead of copies – apply equally to MyGrad and its tensors. After all, MyGrad operates almost entirely in NumPy arrays and NumPy functions under the hood.

3.5.1 Suspending Graph-Tracking for Automatic Differentiation

no_autodiff	Serves as a context manager and decorator for suspending all computational graph tracking.

mygrad.no_autodiff

`mygrad.no_autodiff = <mygrad._utils.graph_tracking._NoAutoDiff object>`

Serves as a context manager and decorator for suspending all computational graph tracking.

Note that memory guarding does not occur in the *no_autodiff* context, so there is no need to nest this context with *mem_guard_off*.

Examples

Demonstrating `no_autodiff` as a context-manager

```
>>> import mygrad as mg
>>> with mg.no_autodiff:
>>>     # all computational graph tracking is suspended
>>>     # within the context
>>>     x = mg.arange(4.)
>>>     (4 * x).backward()  # no autodiff will occur
>>> x.grad is None
```

Demonstrating `no_autodiff` as a decorator

```
>>> @mg.no_autodiff
... def func():
...     # No graph-tracking will occur within
...     # the body of this function
...     pass
```

In the case that you want to run a computation involving MyGrad tensors, but you don't need to access their gradients (e.g. when measuring the "test-time" performance of a model that you are training), then you can use the provided decorator/context-manager for suspending all of MyGrad's "graph-tracking" features.

```
>>> import mygrad as mg
>>> with mg.no_autodiff:
...     # any mygrad code in this context will run faster
...     # but will not produce any gradients
```

Note that this also suspends all memory-guarding (see below), since MyGrad doesn't need to ensure the preservation of any state.

Suspending all graph-tracking features can speed up code involving many small tensors substantially - about a 3x speedup.

3.5.2 Controlling Memory-Guarding Behavior

mem_guard_off	A context manager used to suspend memory-locking behavior
mem_guard_on	A context manager used to enable memory-locking behavior
turn_memory_guarding_off()	Globally disables all memory-guarding mechanisms, except for in contexts where they are explicitly enabled.
turn_memory_guarding_off()	Globally disables all memory-guarding mechanisms, except for in contexts where they are explicitly enabled.

mygrad.mem_guard_off

`mygrad.mem_guard_off = <mygrad._utils.lock_management._NoMemGuard object>`

A context manager used to suspend memory-locking behavior

Examples

```
>>> from mygrad import  mem_guard_off
>>> with mem_guard_off:
...     # array-memory locking is turned off
...     pass
... # previous memory-locking behavior is restored
```

This can also be used as a decorator

```
>>> @mem_guard_off
>>> def f():
...     # array-memory locking is turned off within function
...     return
```

mygrad.mem_guard_on

`mygrad.mem_guard_on = <mygrad._utils.lock_management._WithMemGuard object>`

A context manager used to enable memory-locking behavior

Examples

```
>>> from mygrad import mem_guard_on
>>> with mem_guard_on:
...     # array-memory locking is turned on
...     pass
... # previous memory-locking behavior is restored
```

This can also be used as a decorator

```
>>> @mem_guard_on
>>> def f():
...     # array-memory locking is turned on within function
...     return
```

mygrad.turn_memory_guarding_off

`mygrad.turn_memory_guarding_off()`

Globally disables all memory-guarding mechanisms, except for in contexts where they are explicitly enabled.

See also:

`turn_memory_guarding_on`
Globally enables all memory-guarding mechanisms

`mem_guard_off`
context manager & decorator for suspending memory guarding

`mem_guard_on`
context manager & decorator for enabling memory guarding

Notes

With memory guarding disabled, arrays participating in active computational graphs are not protected from being mutated by the user. Mutating such an array will corrupt the derivatives that are computed via back-propagation, and will produce incorrect results.

This can speed up computations involving many small tensors substantially.

If you want to disable memory guarding at the system level, you can set the system environment variable MY-GRAD_MEM_GUARD=False. NOTE THAT THIS IS NOT RECOMMENDED.

Examples

The following demonstrates how one can unwittingly corrupt backpropagation through a computational graph

```
>>> import mygrad as mg
>>> import numpy as np
>>> mg.turn_memory_guarding_off()  # speeds up calculations, but with risks␣
↪involved..
>>> x = np.arange(3.)
>>> y = mg.ones_like(x)
>>> z = x * y
>>> x[:] = 0  # mutates x, corrupting state associated with z
>>> z.backward()
>>> y.grad  # would be array([0., 1., 2.]) if graph wasn't corrupted
array([0., 0., 0.])
```

By default, MyGrad tracks and locks the readability of all of the NumPy arrays that are involved in computational graphs involving tensors.

These stateful graphs are how MyGrad is able to perform backpropagation and compute the gradients of tensors involved in a given calculation. Because of the stateful nature of a computational graph, mutating a NumPy array inplace could corrupt the state of the computational graph - i.e. the derivatives computed would not accurately reflect the values that were used during the "forward pass". Read the following code to see such a mutation rear its head.

```
>>> import mygrad as mg
>>> import numpy as np
>>> mg.turn_memory_guarding_off()  # speeds up calculations, but with risks involved..
>>> x = np.arange(3.)
>>> y = mg.ones_like(x)
>>> z = x * y
>>> x[:] = 0  # mutates x, corrupting state associated with z
>>> z.backward()
>>> y.grad  # would be array([0., 1., 2.]) if graph wasn't corrupted
array([0., 0., 0.])
```

Note that, were `x` an instance of `Tensor`, there would not be any issue with the above calculation, since MyGrad can track the in-place update on a tensor. MyGrad cannot, on the otherhand track such operations involving only NumPy arrays

Thus MyGrad prohibits such mutations with its aforementioned "memory guarding" behavior, however it is smart about restoring the writeability of all arrays once they are no longer participating in a computational graph (e.g. backpropagation has been performed through the graph).

```
>>> import mygrad as mg
>>> import numpy as np
>>> x = np.arange(3.)
>>> y = mg.ones_like(x)
>>> z = x * y
>>> try:
...     x[:] = 0  # raises because `x` is made read-only
... except ValueError:
...     pass
>>> z.backward()
>>> y.grad  # correct gradient is computed
```

(continues on next page)

(continued from previous page)

```
array([0., 1., 2.])
>>> x[:] = 0  # the writeability of `x` is restored once backprop is complete
```

This memory-guarding behavior comes at a cost: for computations involving many small tensors (e.g. in an handmade RNN) this can lead to slowdowns of ~50%. Thus MyGrad provides various mechanisms for disabling all such memory-guards. Note, however, for computations involving large tensors (e.g. for typical dense and convolutional neural networks), the overhead associated with the memory-guarding feature is likely negligible compared to the core numerical computations at play.

If one wants to enjoy the optimizations associated with removing memory guarding, it is recommended that you first test your code with the default memory guarding enabled; once you have witnessed that MyGrad didn't raise any errors, you can then proceed to run your code "at scale" with memory-guarding disabled.

3.5.3 Make Use of Views but Avoid Involving them in In-Place Operations

Please refer to the section on views and in-place operations for more details. The upshot is: views of tensors are efficient to create, as they do not involve copying any memory, but performing an in-place operations on a tensor will copy that tensor. Furthermore, performing an in-place operation on a view will lead to the creation of a copy of its associated base tensor.

If you are relying on this mutation propagating to many various views, then this can still be a net-gain in performance compared to updating all of them "manually". But, generally, in-place updates on tensors do not have the same performance benefits as do augmentations on NumPy arrays.

3.6 Writing Your Own Operations

Let's write our own "multiply" operation. There are two components to doing this:

- Defining an operation class (a subclass of *Operation*)
- Writing a function that ultimately calls `mygrad.execute_op(YourOp, ...)`

```python
import numpy as np

import mygrad as mg
from mygrad import execute_op
from mygrad.operation_base import Operation
from mygrad.typing import ArrayLike

# All operations should inherit from Operation, or one of its subclasses
class CustomMultiply(Operation):
    """ Performs f(x, y) = x * y """

    def __call__(self, x: mg.Tensor, y: mg.Tensor) -> np.ndarray:
        # This method defines the "forward pass" of the operation.
        # It must bind the variable tensors to the op and compute
        # the output of the operation as a numpy array

        # All tensors must be bound as a tuple to the `variables`
        # instance variable.
        self.variables = (x, y)
```

(continues on next page)

(continued from previous page)

~~~
        # The forward pass should be performed using numpy arrays,
        # not the tensors themselves.
        x_arr = x.data
        y_arr = y.data
        return x_arr * y_arr

    def backward_var(self, grad, index, **kwargs):
        """Given ``grad = d/df``, computes ``/x`` and ``/y``

        ```` is assumed to be the terminal node from which ``.backward()`` was
        called.

        Parameters
        ----------
        grad : numpy.ndarray
            The back-propagated total derivative with respect to the present
            operation: d/df. This will have the same shape as f, the result
            of the forward pass.

        index : Literal[0, 1]
            The index-location of ``var`` in ``self.variables``

        Returns
        -------
        numpy.ndarray
            /x_{i}

        Raises
        ------
        SkipGradient"""
        x, y = self.variables
        x_arr = x.data
        y_arr = y.data

        # The operation need not incorporate specialized logic for
        # broadcasting. The appropriate sum-reductions will be performed
        # by MyGrad's autodiff system.
        if index == 0:  # backprop through a
            return grad * y.data  # /x = (/f)(f/x)
        elif index == 1:  # backprop through b
            return grad * x.data  # /y = (/f)(f/y)


# Our function stitches together our operation class with the
# operation arguments via `mygrad.prepare_op`
def custom_multiply(x: ArrayLike, y: ArrayLike, constant=None) -> mg.Tensor:
    # `execute_op` will take care of:
    #  - casting `x` and `y` to tensors if they are instead array-likes
    #  - propagating 'constant' status to the resulting output based on the inputs
    #  - handling in-place operations (specified via the `out` parameter)
    return execute_op(CustomMultiply, x, y, constant=constant)
~~~

We can now use our differentiable function!

```
>>> x = mg.tensor(2.0)
>>> y = mg.tensor([1.0, 2.0, 3.0])

>>> custom_multiply(x, y).backward()
>>> x.grad, y.grad
(array(6.), array([2., 2., 2.]))
```

3.6.1 Documentation for mygrad.Operation

Operation()	Base class for all tensor operations that support back-propagation of gradients.
Operation.backward(grad, **kwargs)	Back-propagates the gradient through all of the operation's inputs, which are stored in the tuple *self.variables*.
Operation.backward_var(grad, index, **kwargs)	Given `grad = d/df`, computes `/x_{i}`, where `x_{i}` is one of `x1, ...., xn`.

mygrad.operation_base.Operation

class mygrad.operation_base.**Operation**

Base class for all tensor operations that support back-propagation of gradients.

Consider the Operation-instance `f`. A forward-pass through `f` is defined via `f.__call__(...)`. Thus, given tensors `a` and `b`, a computational graph is defined `f.__call__(a, b) -> c`, where the "creator" of tensor `c` is recorded as `f`:

```
(node: a) --+
            -> [operation: f(a, b)] --> (node: c)
(node: b) --+
```

Back-propagating through `c` will instruct `f` to back-propagate the gradient to its inputs, which are recorded as `a` and `b`. Each node then back-propagates to any Operation-instance that is recorded as its creator, and so on.

Methods

`__call__`(*input_vars, **kwargs)	Performs a forward pass, f, of this Operation.
backward(grad, **kwargs)	Back-propagates the gradient through all of the operation's inputs, which are stored in the tuple *self.variables*.
backward_var(grad, index, **kwargs)	Given `grad = d/df`, computes `/x_{i}`, where `x_{i}` is one of `x1, ...., xn`.

mygrad.operation_base.Operation.backward

`Operation.backward`(*grad: ndarray*, ***kwargs*)

Back-propagates the gradient through all of the operation's inputs, which are stored in the tuple *self.variables*.

Constant tensors (*tensor.constant is True*) skipped by this process.

Parameters

grad
[numpy.ndarray] The back-propagated total derivative with respect to the present operation (f): d(out)/df

mygrad.operation_base.Operation.backward_var

abstract `Operation.backward_var`(*grad: ndarray*, *index: int*, ***kwargs*) → ndarray

Given `grad = d/df`, computes `/x_{i}`, where `x_{i}` is one of `x1, ...., xn`.

is assumed to be the terminal node from which `.backward()` was called.

Parameters

grad
[numpy.ndarray] The back-propagated total derivative with respect to the present operation: d/df. This will have the same shape as f, the result of the forward pass.

index
[int] The index-location of `var` in `self.variables`

Returns

numpy.ndarray
/x_{i}

Raises

SkipGradient

grad_post_process_fn	

`__init__`()

Methods

`__init__`()	
`backward`(grad, **kwargs)	Back-propagates the gradient through all of the operation's inputs, which are stored in the tuple *self.variables*.
`backward_var`(grad, index, **kwargs)	Given `grad = d/df`, computes `/x_{i}`, where `x_{i}` is one of `x1, ...., xn`.
`grad_post_process_fn`(grad, var_shape)	

Attributes

`can_return_view`
`variables`

3.7 Tensor creation routines (`mygrad.tensor_creation`)

3.7.1 Array-Like

`tensor`(arr_like[, dtype, constant, copy, ndmin])	Create a tensor
`asarray`(a[, dtype, order])	Convert the input to an array.
`astensor`(t[, dtype, constant])	Convert the input to a tensor.

mygrad.tensor

`mygrad.`**`tensor`**(*arr_like: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, ***, *constant: Optional[bool] = None*, *copy: bool = True*, *ndmin: int = 0*) → Tensor

Create a tensor

This documentation was adapted from that of ``numpy.array`

Parameters

arr_like
[array_like] A tensor, any object exposing the array interface, an object whose __array__ method returns an tensor, a real number, any (nested) sequence.

dtype
[data-type, optional] The desired data-type for the tensor. Restricted to integer and float type. If not specified, then the type will be determined as the minimum type required to hold the objects in the sequence.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant_tensor.grad` will always return `None`).

If a new tensor is returned:

- Defaults to `False` for float-type data.
- Defaults to `True` for integer-type data.

copy
[bool, optional] If true (default), or if a copy is needed to satisfy any of the other requirements (`dtype`, `constant`, etc.) then a new tensor is created from copied data. Otherwise the tensor will be returned unchanged.

ndmin
[int, optional] Specifies the minimum number of dimensions that the resulting tensor should have. Ones will be prepended to the shape as needed to meet this requirement.

Returns

out
[Tensor] A tensor satisfying the specified requirements.

See also:

empty_like
Return an empty tensor with shape and type of input.

ones_like
Return an tensor of ones with shape and type of input.

zeros_like
Return an tensor of zeros with shape and type of input.

full_like
Return a new tensor with shape of input filled with value.

empty
Return a new uninitialized tensor.

ones
Return a new tensor setting values to one.

zeros
Return a new tensor setting values to zero.

full
Return a new tensor of given shape filled with value.

Examples

```
>>> import mygrad as mg
>>> mg.tensor([1, 2, 3])
Tensor([1, 2, 3])
```

Upcasting:

```
>>> mg.tensor([1, 2, 3.0])
Tensor([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> mg.tensor([[1, 2], [3, 4]])
Tensor([[1, 2],
        [3, 4]])
```

Minimum dimensions 2:

```
>>> mg.tensor([1, 2, 3], ndmin=2)
Tensor([[1, 2, 3]])
```

Type provided:

```
>>> mg.tensor([1, 2, 3], dtype="float32")
Tensor([1., 2., 3.], dtype=float32)
```

mygrad.asarray

`mygrad.asarray`(*a: ArrayLike*, *dtype: Optional[DTypeLike] = None*, *order: Optional[str] = None*) → ndarray

Convert the input to an array.

This docstring is adapted from that of `numpy.asarray`

Parameters

a
[array_like] Input data, in any form - including a mygrad tensor - that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

dtype
[data-type, optional] By default, the data-type is inferred from the input data.

order
[{'C', 'F'}, optional] Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

Returns

out
[ndarray] Array interpretation of *a*. No copy is performed if the input is already an ndarray with matching dtype and order. If *a* is a subclass of ndarray, a base class ndarray is returned.

Examples

Convert a list into an array:

```
>>> import mygrad as mg
>>> a = [1, 2]
>>> mg.asarray(a)
array([1, 2])
```

Convert a tensor into an array. No copy of the underlying numpy array is created:

```
>>> t = mg.Tensor([1, 2.])
>>> mg.asarray(t)
array([1., 2.])
>>> t.data is np.asarray(t))
True
```

Existing arrays are not copied:

```
>>> a = np.array([1, 2])
>>> mg.asarray(a) is a
True
```

If *dtype* is set, array is copied only if dtype does not match:

```
>>> a = np.array([1, 2], dtype=np.float32)
>>> mg.asarray(a, dtype=np.float32) is a
True
>>> mg.asarray(a, dtype=np.float64) is a
False
```

Contrary to *asanyarray*, ndarray subclasses are not passed through:

```
>>> issubclass(np.recarray, np.ndarray)
True
>>> a = np.array([(1.0, 2), (3.0, 4)], dtype='f4,i4').view(np.recarray)
>>> mg.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
```

mygrad.astensor

`mygrad.astensor`(*t: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Convert the input to a tensor.

A tensor *t* is returned unchanged - its gradient and computational graph state preserved - if dtype and constant are compatible. A copy of the underlying numpy array is created only if dtype is incompatible or if a non-constant tensor is being created from a constant.

Parameters

t
[array_like] Input data, in any form that can be converted to a tensor. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

dtype
[data-type, optional] By default, the data-type is inferred from the input data.

constant
[Optional[bool]] By default, *constant* is inferred from *t* if *t* is a tensor, otherwise it defaults to *False*.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

out
[Tensor] Tensor interpretation of *a*. No copy is performed if the input is already a tensor with matching dtype and constant-flag.

Examples

Convert a list into an array:

```
>>> import mygrad as mg
>>> import numpy as np
>>> t = [1, 2]
>>> mg.astensor(t)
Tensor([1, 2])
```

Convert an array into a tensor. No copy of the underlying numpy array is created:

```
>>> a = np.array([1.0, 2.0])
>>> mg.astensor(a)
Tensor([1., 2.])
>>> a is mg.astensor(a).data
True
```

Existing tensors are not copied and their gradients and computational graphs are preserved:

```
>>> t1 = 2 * mg.tensor([1, 2])
>>> t2 = mg.astensor(t1)
>>> t1 is t2
True
>>> t1.creator is t2.creator
True
```

If *dtype* is set, a new tensor is created - with copied data - only if dtype does not match:

```
>>> t = mg.Tensor([1, 2], dtype=np.float32)
>>> mg.astensor(t, dtype=np.float32) is t
True
>>> mg.astensor(t, dtype=np.float64) is t
False
```

Otherwise, if *constant* is set, a new tensor is created (with no copy of the underlying data) only if constant doesn't match.

```
>>> t1 = mg.tensor([1, 2], constant=False)
>>> mg.astensor(t1, constant=False) is t
True
>>> mg.astensor(t1, constant=True) is t1
False
>>> mg.astensor(t1, constant=True).data is t1.data
True
```

3.7.2 Ones and zeros

`ones`(shape[, dtype, constant])	Return a Tensor of the given shape and type, filled with ones.
`ones_like`(other[, dtype, shape, constant])	Return a Tensor of the same shape and type as the given, filled with ones.
`zeros`(shape[, dtype, constant])	Return a Tensor of the given shape and type, filled with zeros.
`zeros_like`(other[, dtype, shape, constant])	Return a Tensor of the same shape and type as the given, filled with zeros.
`eye`(N[, M, k, dtype, constant])	Return a 2D Tensor with ones on the diagonal and zeros elsewhere.
`identity`(n[, dtype, constant])	Return the identity Tensor; a square Tensor with 1s on the main diagonal and 0s elsewhere.
`full`(shape, fill_value[, dtype, constant])	Return a Tensor of the given shape and type, filled with *fill_value*.
`full_like`(other, fill_value[, dtype, shape, ...])	Return a Tensor of the same shape and type as the given, filled with *fill_value*.
`empty`(shape[, dtype, constant])	Return a new Tensor of the given shape and type, without initializing entries.
`empty_like`(other[, dtype, shape, constant])	Return a new Tensor of the same shape and type as the given array.

mygrad.ones

`mygrad.ones`(*shape: ~typing.Union[~typing.Sequence[int], int], dtype: ~mygrad.typing._dtype_like.DTypeLikeReals = <class 'numpy.float32'>, *, constant: ~typing.Optional[bool] = None*) → Tensor

Return a Tensor of the given shape and type, filled with ones.

This docstring was adapted from `numpy.ones` [1]

Parameters

shape
[Union[int, Tuple[int]]] The shape of the output Tensor.

dtype
[data-type, optional (default=numpy.float32)] The data type of the output Tensor.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A Tensor of ones with the given shape and data type.

See also:

`ones_like`
Return an tensor of ones with shape and type of input.

empty
Return a new uninitialized tensor.

zeros
Return a new tensor setting values to zero.

full
Return a new tensor of given shape filled with value.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.ones(5)
Tensor([ 1.,  1.,  1.,  1.,  1.])
```

```
>>> mg.ones((5,), dtype=int)
Tensor([1, 1, 1, 1, 1])
```

```
>>> mg.ones((2, 1))
Tensor([[ 1.],
       [ 1.]])
```

```
>>> mg.ones((2, 2))
Tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

mygrad.ones_like

`mygrad.ones_like`(*other: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, *shape: Optional[Union[int, Sequence[int]]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor of the same shape and type as the given, filled with ones.

This docstring was adapted from `numpy.ones_like` [1]

Parameters

other
[array_like] The Tensor or array whose shape and datatype should be mirrored.

dtype
[Optional[DTypeLikeReals]] Override the data type of the returned Tensor with this value, or None to not override.

shape
[Optional[Union[int, Sequence[int]]]] If specified, overrides the shape of the result

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation. If `None` then:

Inferred from `other`, if other is a tensor Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Returns

Tensor
A Tensor of ones whose shape and data type match *other*.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.arange(6).reshape((2, 3))
>>> x
Tensor([[0, 1, 2],
        [3, 4, 5]])
```

```
>>> mg.ones_like(x)
Tensor([[1, 1, 1],
        [1, 1, 1]])
```

```
>>> y = mg.arange(3, dtype=float)
>>> y
Tensor([ 0.,  1.,  2.])
```

```
>>> mg.ones_like(y)
Tensor([ 1.,  1.,  1.])
```

mygrad.zeros

`mygrad.zeros`(*shape: ~typing.Union[~typing.Sequence[int], int], dtype: ~mygrad.typing._dtype_like.DTypeLikeReals = <class 'numpy.float32'>, *, constant: ~typing.Optional[bool] = None*) → Tensor

Return a Tensor of the given shape and type, filled with zeros.

This docstring was adapted from `numpy.zeros` [1]

Parameters

shape
[Union[int, Tuple[int]]] The shape of the output Tensor.

dtype
[data-type, optional (default=numpy.float32)] The data type of the output Tensor.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A Tensor of zeros with the given shape and data type.

See also:

ones_like
Return an tensor of ones with shape and type of input.

empty
Return a new uninitialized tensor.

ones
Return a new tensor setting values to one.

full
Return a new tensor of given shape filled with value.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.zeros(5)
Tensor([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> mg.zeros((5,), dtype=int, constant=True) # tensor will not back-propagate a␣
↪gradient
Tensor([0, 0, 0, 0, 0])
```

```
>>> mg.zeros((2, 1))
Tensor([[ 0.],
        [ 0.]])
```

```
>>> mg.zeros((2, 2))
Tensor([[ 0.,  0.],
        [ 0.,  0.]])
```

mygrad.zeros_like

`mygrad.zeros_like`(*other: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, *shape: Optional[Union[int, Sequence[int]]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor of the same shape and type as the given, filled with zeros.

This docstring was adapted from `numpy.zeros_like` [1]

Parameters

other
[ArrayLike] The Tensor or array whose shape and datatype should be mirrored.

dtype
[Optional[DTypeLikeReals]] Override the data type of the returned Tensor with this value, or None to not override.

shape
[Optional[int, Sequence[int]]] If specified, overrides the shape of the result

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation. If `None` then:

Inferred from `other`, if other is a tensor Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A Tensor of zeros whose shape and data type match *other*.

See also:

`empty_like`
Return an empty tensor with shape and type of input.

`ones_like`
Return an tensor of ones with shape and type of input.

`full_like`
Return a new tensor with shape of input filled with value.

`zeros`
Return a new tensor setting values to zero.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.arange(6).reshape((2, 3))
>>> x
Tensor([[0, 1, 2],
        [3, 4, 5]])
```

```
>>> mg.zeros_like(x, constant=True)  # tensor will not back-propagate a gradient
Tensor([[0, 0, 0],
        [0, 0, 0]])
```

```
>>> y = mg.arange(3, dtype=float)
>>> y
Tensor([ 0.,  1.,  2.])
```

```
>>> mg.zeros_like(y)
Tensor([ 0.,  0.,  0.])
```

mygrad.eye

`mygrad.eye`(*N: int*, *M: ~typing.Optional[int] = None*, *k: int = 0*, *dtype: ~mygrad.typing._dtype_like.DTypeLikeReals = <class 'float'>*, ***, *constant: ~typing.Optional[bool] = None*) → Tensor

Return a 2D Tensor with ones on the diagonal and zeros elsewhere.

This docstring was adapted from `numpy.eye` [1]

Parameters

N
[int] The number of rows in the output Tensor.

M
[int, optional (default=None)] The number of columns in the output, or None to match *rows*.

k
[int, optional (default=0)] The index of the diagonal. 0 is the main diagonal; a positive value is the upper diagonal, while a negative value refers to the lower diagonal.

dtype
[data-type, optional (default=numpy.float32)] The data type of the output Tensor.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A tensor whose elements are 0, except for the k-th diagonal, whose values are 1.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.eye(2, dtype=int)
Tensor([[1, 0],
        [0, 1]])
>>> mg.eye(3, k=1)
Tensor([[ 0.,  1.,  0.],
        [ 0.,  0.,  1.],
        [ 0.,  0.,  0.]])
```

mygrad.identity

`mygrad.identity`(*n: int*, *dtype: ~mygrad.typing._dtype_like.DTypeLikeReals = <class 'float'>*, ***, *constant: ~typing.Optional[bool] = None*) → Tensor

Return the identity Tensor; a square Tensor with 1s on the main diagonal and 0s elsewhere.

This docstring was adapted from `numpy.identity` [1]

Parameters

n
[int] The number of rows and columns in the output Tensor.

dtype
[data-type, optional (default=numpy.float32)] The data type of the output Tensor.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A square Tensor whose main diagonal is 1 and all other elements are 0.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.identity(3)
Tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

mygrad.full

`mygrad.full`(*shape: Union[Sequence[int], int]*, *fill_value: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor of the given shape and type, filled with *fill_value*.

This docstring was adapted from `numpy.full` [1]

Parameters

shape
[Union[int, Iterable[int]]] The shape of the output Tensor.

fill_value
[ArrayLike] The value with which to fill the output Tensor. Note that this function is not differentiable – the resulting tensor will not backprop through *fill_value*.

The value with which to fill the output Tensor.

dtype
[Optional[DTypeLikeReals]] The data type of the output Tensor, or None to match *fill_value*..

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A Tensor of *fill_value* with the given shape and dtype.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.full((2, 2), 33)
Tensor([[ 33,  33],
        [ 33,  33]])
```

```
>>> mg.full((2, 2), 10)
Tensor([[10, 10],
        [10, 10]])
```

mygrad.full_like

`mygrad.full_like`(*other: ArrayLike*, *fill_value: Union[int, float]*, *dtype: Optional[DTypeLikeReals] = None*, *shape: Optional[Union[int, Sequence[int]]] = None*, *constant: Optional[bool] = None*) → Tensor

Return a Tensor of the same shape and type as the given, filled with *fill_value*.

This docstring was adapted from `numpy.full_like` [1]

Parameters

other
[ArrayLike] The tensor or array whose shape and datatype should be mirrored.

fill_value
[Real] The value with which to fill the output Tensor.

dtype
[Optional[DTypeLikeReals]] Override the data type of the returned Tensor with this value, or None to not override.

shape
[Optional[int, Sequence[int]]] If specified, overrides the shape of the result

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation. If `None` then:

Inferred from `other`, if other is a tensor Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Returns

Tensor
A Tensor of *fill_value* whose shape and data type match *other*.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.arange(6, dtype=int)
>>> mg.full_like(x, 1)
Tensor([1, 1, 1, 1, 1, 1])
>>> mg.full_like(x, 0.1)
Tensor([0, 0, 0, 0, 0, 0])
>>> mg.full_like(x, 0.1, dtype=np.double)
Tensor([ 0.1,  0.1,  0.1,  0.1,  0.1,  0.1])
>>> mg.full_like(x, np.nan, dtype=np.double)
Tensor([ nan,  nan,  nan,  nan,  nan,  nan])
```

```
>>> y = mg.arange(6, dtype=np.double)
>>> mg.full_like(y, 0.1)
Tensor([ 0.1,  0.1,  0.1,  0.1,  0.1,  0.1])
```

mygrad.empty

`mygrad.empty`(*shape: ~typing.Union[~typing.Sequence[int], int], dtype: ~mygrad.typing._dtype_like.DTypeLikeReals = <class 'numpy.float32'>, *, constant: ~typing.Optional[bool] = None*) → Tensor

Return a new Tensor of the given shape and type, without initializing entries.

This docstring was adapted from `numpy.empty` [1]

Parameters

shape
[Union[int, Tuple[int]]] The shape of the empty array.

dtype
[data-type, optional (default=numpy.float32)] The data type of the output Tensor.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A tensor of uninitialized data of the given shape and dtype.

See also:

empty_like
Return an empty tensor with shape and type of input.

ones
Return a new tensor setting values to one.

zeros
Return a new tensor setting values to zero.

full
Return a new tensor of given shape filled with value.

Notes

empty, unlike *zeros*, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.empty([2, 2], constant=True)
Tensor([[ -9.74499359e+001,   6.69583040e-309],
        [  2.13182611e-314,   3.06959433e-309]])         #random
```

```
>>> mg.empty([2, 2], dtype=int)
Tensor([[-1073741821, -1067949133],
        [  496041986,    19249760]])                     #random
```

mygrad.empty_like

mygrad.**empty_like**(*other: ArrayLike*, *dtype: Optional[DTypeLikeReals] = None*, *shape: Optional[Union[int, Sequence[int]]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return a new Tensor of the same shape and type as the given array.

This docstring was adapted from `numpy.empty_like` [1]

Parameters

other
[ArrayLike] The Tensor or array whose shape and datatype should be mirrored.

dtype
[Optional[DTypeLikeReals]] Override the data type of the returned Tensor with this value, or None to not override.

shape
[Optional[Union[int, Sequence[int]]]] If specified, overrides the shape of the result

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation. If `None` then:

Inferred from `other`, if other is a tensor Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Returns

Tensor
A tensor of uninitialized data whose shape and type match *other*.

See also:

`empty`
Return a new Tensor of the given shape and type, without initializing entries.

`ones`
Return a new tensor setting values to one.

`zeros`
Return a new tensor setting values to zero.

`full`
Return a new tensor of given shape filled with value.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.arange(4).reshape(2, 2)
>>> mg.empty_like(x, constant=True)
Tensor([[ -9.74499359e+001,   6.69583040e-309],
        [  2.13182611e-314,   3.06959433e-309]])         #random
```

```
>>> mg.empty_like(x, dtype=int)
Tensor([[-1073741821, -1067949133],
        [  496041986,    19249760]])                     #random
```

3.7.3 Numerical ranges

`arange`([start,] stop[, step,][, dtype, constant])	Return a Tensor with evenly-spaced values within a given interval.
`linspace`(start, stop[, num, endpoint, ...])	Return a Tensor with evenly-spaced numbers over a specified interval.
`logspace`(start, stop[, num, endpoint, base, ...])	Return a Tensor with evenly-spaced numbers over a specified interval on a log scale.
`geomspace`(start, stop[, num, endpoint, ...])	Return a Tensor with evenly-spaced values in a geometric progression.

mygrad.arange

`mygrad.`**`arange`**`(`[*start*], *stop*[, *step*], *dtype=None*, ***, *constant=None*)

Return a Tensor with evenly-spaced values within a given interval.

Values are generated within [start, stop). Note that for non-integer steps, results may be inconsistent; you are better off using *linspace* instead.

This docstring was adapted from `numpy.arange` [1]

Parameters

start
[Real, optional, default=0] The start of the interval, inclusive.

stop
[Real] The end of the interval, exclusive.

step
[int, optional (default=1)] The spacing between successive values.

dtype
[Optional[DTypeLikeReals]] The data type of the output Tensor, or None to infer from the inputs.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
A Tensor of evenly-spaced values in [start, end).

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.arange(3)
Tensor([0, 1, 2])
>>> mg.arange(3.0, constant=True)  # resulting tensor will not back-propagate a␣
↪gradient
Tensor([ 0.,  1.,  2.])
>>> mg.arange(3,7)
Tensor([3, 4, 5, 6])
>>> mg.arange(3,7,2)
Tensor([3, 5])
```

mygrad.linspace

mygrad.**linspace**(*start: ArrayLike*, *stop: ArrayLike*, *num: int = 50*, *endpoint: bool = True*, *dtype: Optional[DTypeLikeReals] = None*, *axis: int = 0*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor with evenly-spaced numbers over a specified interval.

Values are generated within [start, stop], with the endpoint optionally excluded.

This docstring was adapted from `numpy.linspace` [1]

Parameters

start
[ArrayLike] The starting value of the sequence, inclusive.

stop
[ArrayLike] The ending value of the sequence, inclusive unless *include_endpoint* is False.

num
[int, optional (default=50)] The number of values to generate. Must be non-negative.

endpoint
[bool, optional (default=True)] Whether to include the endpoint in the Tensor. Note that if False, the step size changes to accommodate the sequence excluding the endpoint.

dtype
[Optional[DTypeLikeReals]] The data type of the output Tensor, or None to infer from the inputs.

axis
[int, optional (default=0)] The axis in the result to store the samples - for array-like start/stop.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor

See also:

arange
Similar to *linspace*, but uses a step size (instead of the number of samples).

logspace
Samples uniformly distributed in log space.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.linspace(2.0, 3.0, num=5)
Tensor([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
>>> mg.linspace(2.0, 3.0, num=5, endpoint=False)
Tensor([ 2. ,  2.2,  2.4,  2.6,  2.8])
```

mygrad.logspace

mygrad.**logspace**(*start: ArrayLike*, *stop: ArrayLike*, *num: int = 50*, *endpoint: bool = True*, *base: Union[int, float] = 10*, *dtype: Optional[DTypeLikeReals] = None*, *axis: int = 0*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor with evenly-spaced numbers over a specified interval on a log scale. This is not a differentiable function - it does not propagate gradients to its inputs.

In linear space, values are generated within [base**start, base**stop], with the endpoint optionally excluded.

This docstring was adapted from `numpy.logspace` [1]

Parameters

start
[ArrayLike] The starting value of the sequence, inclusive; start at *base ** start*.

stop
[ArrayLike] The ending value of the sequence, inclusive unless *include_endpoint* is False; end at *base ** stop*.

num
[int, optional (default=50)] The number of values to generate. Must be non-negative.

endpoint
[bool, optional (default=True)] Whether to include the endpoint in the Tensor. Note that if False, the step size changes to accommodate the sequence excluding the endpoint.

base
[Real, optional (default=10)] The base of the log space.

dtype
[Optional[DTypeLikeReals]] The data type of the output Tensor, or None to infer from the inputs.

axis
[int, optional (default=0)] The axis in the result to store the samples - for array-like start/stop.

constant
[Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor

See also:

`arange`
Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

`linspace`
Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

`geomspace`
Similar to logspace, but with endpoints specified directly.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.logspace(2.0, 3.0, num=4)
Tensor([  100.        ,   215.443469  ,   464.15888336,  1000.        ])
>>> mg.logspace(2.0, 3.0, num=4, endpoint=False)
Tensor([ 100.        ,  177.827941  ,  316.22776602,  562.34132519])
>>> mg.logspace(2.0, 3.0, num=4, base=2.0)
Tensor([ 4.        ,  5.0396842 ,  6.34960421,  8.        ])
```

mygrad.geomspace

`mygrad.geomspace`(*start: ArrayLike*, *stop: ArrayLike*, *num=50*, *endpoint=True*, *dtype=None*, *axis=0*, ***, *constant: Optional[bool] = None*) → Tensor

Return a Tensor with evenly-spaced values in a geometric progression.

Each output sample is a constant multiple of the previous output.

This docstring was adapted from `numpy.geomspace` [1]

Parameters

start
: [ArrayLike] The starting value of the output.

stop
: [ArrayLike] The ending value of the sequence, inclusive unless *endpoint* is false.

num
: [int, optional (default=50)] The number of values to generate. Must be non-negative.

endpoint
: [bool, optional (default=True)] Whether to include the endpoint in the Tensor. Note that if False, the step size changes to accommodate the sequence excluding the endpoint.

dtype
: [Optional[DTypeLikeReals]] The data type of the output Tensor, or None to infer from the inputs.

axis
: [int, optional (default=0)] The axis in the result to store the samples - for array-like start/stop.

constant
: [Optional[bool]] If `True`, this tensor is a constant, and thus does not facilitate back propagation.

  Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

  Integer-type tensors must be constant.

Returns

Tensor

See also:

`logspace`
: Similar to geomspace, but with endpoints specified using log and base.

`linspace`
: Similar to geomspace, but with arithmetic instead of geometric progression.

`arange`
: Similar to linspace, with the step size specified instead of the number of samples.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.geomspace(1, 1000, num=4)
Tensor([    1.,    10.,   100.,  1000.])
>>> mg.geomspace(1, 1000, num=3, endpoint=False)
Tensor([   1.,   10.,  100.])
>>> mg.geomspace(1, 1000, num=4, endpoint=False)
Tensor([   1.        ,    5.62341325,   31.6227766 ,  177.827941  ])
>>> mg.geomspace(1, 256, num=9)
Tensor([   1.,    2.,    4.,    8.,   16.,   32.,   64.,  128.,  256.])
```

Note that the above may not produce exact integers:

```
>>> mg.geomspace(1, 256, num=9, dtype=int)
Tensor([  1,   2,   4,   7,  16,  32,  63, 127, 256])
>>> np.around(mg.geomspace(1, 256, num=9).data).astype(int)
array([  1,   2,   4,   8,  16,  32,  64, 128, 256])
```

Negative, and decreasing inputs are allowed:

```
>>> mg.geomspace(1000, 1, num=4)
Tensor([ 1000.,   100.,    10.,     1.])
>>> mg.geomspace(-1000, -1, num=4)
Tensor([-1000.,  -100.,   -10.,    -1.])
```

3.8 Tensor manipulation routines (`mygrad.tensor_manip`)

3.8.1 Changing array shape

ravel(a, *[, constant])	Flattens contents of a tensor into a contiguous 1-D array.
reshape(a, newshape, *[, constant])	Returns a tensor with a new shape, without changing its data.
Tensor.flatten(*[, constant])	Return a copy of the tensor collapsed into one dimension.

mygrad.ravel

`mygrad.`**`ravel`**(*a: ArrayLike*, *, *constant: Optional[bool] = None*) → Tensor

Flattens contents of a tensor into a contiguous 1-D array. A copy is made only if needed.

This docstring was adapted from `numpy.ravel`.

Parameters

a
[ArrayLike] The tensor to be flattened

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Notes

`ravel` utilizes C-ordering, meaning that it reads & writes elements using C-like index ordering; the last axis index changing fastest, and, proceeding in reverse order, the first axis index changing slowest.

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([[1, 2],
...                [3, 4]])
>>> mg.ravel(x)
Tensor([1, 2, 3, 4])
```

mygrad.reshape

`mygrad.reshape`(*a: ArrayLike*, *newshape: Union[int, Shape]*, *, *constant: Optional[bool] = None*) → Tensor

Returns a tensor with a new shape, without changing its data.

This docstring was adapted from `numpy.reshape`

Parameters

a
[ArrayLike] The tensor to be reshaped

newshape
[Union[int, Tuple[int, . . .]]] The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D tensor of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the tensor and remaining dimensions.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor
`a` with its shape changed permuted. A new tensor is returned.

Notes

`reshape` utilizes C-ordering, meaning that it reads & writes elements using C-like index ordering; the last axis index changing fastest, and, proceeding in reverse order, the first axis index changing slowest.

Examples

```
>>> import mygrad as mg
>>> a = mg.Tensor([[1,2,3], [4,5,6]])
>>> mg.reshape(a, 6)
Tensor([1, 2, 3, 4, 5, 6])
```

```
>>> mg.reshape(a, (3,-1))   # the unspecified value is inferred to be 2
Tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

mygrad.Tensor.flatten

`Tensor.flatten`(**, constant: Optional[bool] = None*) → Tensor

Return a copy of the tensor collapsed into one dimension.

This docstring was adapted from `numpy.ndarray.flatten`.

Parameters

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor
A copy of the input tensor, flattened to one dimension.

Notes

To return a flattened view of the tensor, use `x.reshape(-1)`.

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([[1, 2],
...                [3, 4]])
>>> x.flatten()
Tensor([1, 2, 3, 4])
```

3.8.2 Transpose-like operations

moveaxis(a, source, destination, *[, constant])	Move axes of a tensor to new positions.
roll(a, shift[, axis, constant])	Roll tensor elements along a given axis.
swapaxes(a, axis1, axis2, *[, constant])	Interchange two axes of a tensor.
Tensor.T	Same as self.transpose(), except that self is returned if self.ndim < 2 and a view of the underlying data is utilized whenever possible.
transpose(a, *axes[, constant])	Permute the dimensions of a tensor.

mygrad.moveaxis

`mygrad.moveaxis`(*a: ArrayLike*, *source: Union[int, Tuple[int, ...]]*, *destination: Union[int, Tuple[int, ...]]*, *, *constant: Optional[bool] = None*) → Tensor

Move axes of a tensor to new positions. Other axes remain in their original order.

Parameters

a
[ArrayLike] The array whose axes should be reordered.

source
[Union[int, Sequence[int]]] Original positions of the axes to move. These must be unique.

destination
[Union[int, Sequence[int]]] Destination positions for each of the original axes. These must also be unique.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns
——-
result
[mygrad.Tensor] Array with moved axes. This array is a view of the input array..

Examples

```
>>> from mygrad import zeros, moveaxis
>>> x = zeros((3, 4, 5))
>>> moveaxis(x, 0, -1).shape
(4, 5, 3)
>>> moveaxis(x, -1, 0).shape
(5, 3, 4)
>>> moveaxis(x, [0, 1], [-1, -2]).shape
(5, 4, 3)
```

mygrad.roll

`mygrad.roll`(*a: ArrayLike*, *shift: Union[int, Tuple[int, ...]]*, *axis=None*, *, *constant: Optional[bool] = None*) → Tensor

Roll tensor elements along a given axis.

Elements that roll beyond the end of an axis “wrap back around” to the beginning.

This docstring was adapted from `numpy.roll`

Parameters

a
[ArrayLike] Input tensor.

shift
[Union[int, Tuple[int, . . .]]] The number of places by which elements are shifted. If a tuple, then *axis* must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while *axis* is a tuple of ints, then the same value is used for all given axes.

axis
[Optional[Union[int, Tuple[int, . . .]]]] Axis or axes along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

res
[Tensor] Output array, with the same shape as *a*.

Examples

```
>>> import mygrad as mg
>>> x = mg.arange(10)
>>> mg.roll(x, 2)
Tensor([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
>>> x2 = mg.reshape(x, (2,5))
>>> x2
Tensor([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> mg.roll(x2, 1)
Tensor([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])
>>> mg.roll(x2, 1, axis=0)
Tensor([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])
>>> mg.roll(x2, 1, axis=1)
Tensor([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])
```

mygrad.swapaxes

`mygrad.swapaxes`(*a: ArrayLike*, *axis1: int*, *axis2: int*, ***, *constant: Optional[bool] = None*) → Tensor

Interchange two axes of a tensor.

Parameters

a
[ArrayLike] Input array.

axis1
[int] First axis.

axis2
[int] Second axis.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

Examples

```
>>> from mygrad import Tensor, swapaxes
>>> x = Tensor([[1, 2, 3]])
>>> swapaxes(x, 0, 1)
Tensor([[1],
       [2],
       [3]])
>>> x = Tensor([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])
>>> x
Tensor([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> swapaxes(x, 0, 2)
Tensor([[[0, 4],
        [2, 6]],
       [[1, 5],
        [3, 7]]])
```

mygrad.transpose

`mygrad.`**`transpose`**(*a: ArrayLike*, **axes: int*, *constant: Optional[bool] = None*) → Tensor
Permute the dimensions of a tensor.

Parameters

a
[ArrayLike] The tensor to be transposed

axes
[int] By default, reverse the dimensions, otherwise permute the axes according to the values given.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor
a with its axes permuted. A new tensor is returned.

Examples

```
>>> import mygrad as mg
>>> a = mg.tensor([[1, 2], [3, 4]])
>>> a
Tensor([[1, 2],
        [3, 4]])
>>> a.transpose()
Tensor([[1, 3],
        [2, 4]])
>>> a.transpose((1, 0))
Tensor([[1, 3],
        [2, 4]])
>>> a.transpose(1, 0)
Tensor([[1, 3],
        [2, 4]])
```

3.8.3 Changing number of dimensions

atleast_1d()	Convert inputs to tensors with at least one dimension.
atleast_2d()	Convert inputs to tensors with at least one dimension.
atleast_3d()	Convert inputs to tensors with at least one dimension.
broadcast_to(a, shape, *[, constant])	Broadcast a tensor to a new shape.
expand_dims(a, axis, *[, constant])	Expand the dimensions of a tensor by adding a new axis.
squeeze(a[, axis, constant])	Remove single-dimensional entries from the shape of a tensor.

mygrad.atleast_1d

mygrad.**atleast_1d**(*tensors: ArrayLike*, *, *constant: Optional[bool] = None*) → Tensor

mygrad.**atleast_1d**(**tensors: ArrayLike*, *constant: Optional[bool] = None*) → List[Tensor]

Convert inputs to tensors with at least one dimension.

Scalar inputs are converted to 1-dimensional tensors, whilst higher-dimensional inputs are preserved.

This docstring was adapted from `numpy.atleast_1d`.

Parameters

tens1, tens2, …
[ArrayLike] One or more input tensors.

Returns

ret
[Tensor | List[Tensor]] A tensor, or list of tensors, each with `a.ndim >= 1`. Copies are made only if necessary.

See also:

atleast_2d, *atleast_3d*

Examples

```
>>> import mygrad as mg
>>> mg.atleast_1d(1.0)
array([1.])
```

```
>>> x = mg.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
Tensor([[0., 1., 2.],
        [3., 4., 5.],
        [6., 7., 8.]])
>>> mg.atleast_1d(x) is x
True
```

```
>>> mg.atleast_1d(1, [3, 4])
[Tensor([1]), Tensor([3, 4])]
```

`numpy.atleast_1d` will dispatch appropriately on tensors.

```
>>> x = mg.tensor(2.)
>>> np.atleast_1d(x)
Tensor([2.])
```

```
>>> np.atleast_1d(x).backward()
>>> x.grad
array(1.)
```

If any argument to `numpy.atleast_1d` is a Tensor, `mygrad.atleast_1d` will be dispatched on all of the arguments.

```
>>> np.atleast_1d(x, 1.)
[Tensor([2.]), Tensor([1.])]
```

mygrad.atleast_2d

`mygrad.`**`atleast_2d`**(*tensors: ArrayLike*, *, *constant: Optional[bool] = None*) → Tensor

`mygrad.`**`atleast_2d`**(**tensors: ArrayLike*, *constant: Optional[bool] = None*) → List[Tensor]

Convert inputs to tensors with at least one dimension.

Scalar inputs are converted to 2-dimensional tensors, whilst higher-dimensional inputs are preserved.

This docstring was adapted from `numpy.atleast_2d`.

Parameters

tens1, tens2, ...
[ArrayLike] One or more input tensors.

Returns

ret
[Tensor | List[Tensor]] A tensor, or list of tensors, each with `a.ndim >= 2`. Copies are made only if necessary.

See also:

atleast_1d, *atleast_3d*

Examples

```
>>> import mygrad as mg
>>> mg.atleast_2d(3.0)
Tensor([[3.]])
```

```
>>> x = mg.arange(3.0)
>>> mg.atleast_2d(x)
array([[0., 1., 2.]])
>>> mg.atleast_2d(x).base is x
True
```

```
>>> mg.atleast_2d(1, [1, 2], [[1, 2]])
[Tensor([[1]]), Tensor([[1, 2]]), Tensor([[1, 2]])]
```

`numpy.atleast_2d` will dispatch appropriately on tensors.

```
>>> x = mg.tensor(2.)
>>> np.atleast_2d(x)
Tensor([[2.]])
```

```
>>> np.atleast_2d(x).backward()
>>> x.grad
array(1.)
```

If any argument to `numpy.atleast_2d` is a Tensor, `mygrad.atleast_2d` will be dispatched on all of the arguments.

```
>>> np.atleast_2d(x, 1.)
[Tensor([[2.]]), Tensor([[1.]])]
```

mygrad.atleast_3d

`mygrad.atleast_3d`(*tensors: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

`mygrad.atleast_3d`(**tensors: ArrayLike*, *constant: Optional[bool] = None*) → List[Tensor]

Convert inputs to tensors with at least one dimension.

Scalar inputs are converted to 3-dimensional tensors, whilst higher-dimensional inputs are preserved.

This docstring was adapted from `numpy.atleast_3d`.

Parameters

tens1, tens2, …
[ArrayLike] One or more input tensors.

Returns

ret
: [Tensor | List[Tensor]] A tensor, or list of tensors, each with `a.ndim >= 3`. Copies are made only if necessary. For example, a 1-D tensor of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D tensor of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

See also:

atleast_1d, *atleast_3d*

Examples

```
>>> import mygrad as mg
>>> mg.atleast_3d(3.0)
Tensor([[[3.]]])
```

```
>>> x = mg.arange(3.0)
>>> mg.atleast_3d(x).shape
(1, 3, 1)
>>> mg.atleast_3d(x).base is x
True
```

```
>>> x = mg.arange(12.0).reshape(4,3)
>>> mg.atleast_3d(x).shape
(4, 3, 1)
```

```
>>> mg.atleast_3d(1, [[1, 2]], [[[[1, 2]]]])
[Tensor([[[1]]]), Tensor([[[1, 2]]]), Tensor([[[[1, 2]]]])]
```

`numpy.atleast_3d` will dispatch appropriately on tensors.

```
>>> x = mg.tensor(2.)
>>> np.atleast_3d(x)
Tensor([[[2.]]])
```

```
>>> np.atleast_3d(x).backward()
>>> x.grad
array(1.)
```

If any argument to `numpy.atleast_3d` is a Tensor, `mygrad.atleast_3d` will be dispatched on all of the arguments.

```
>>> np.atleast_3d(x, 1.)
[Tensor([[[2.]]]), Tensor([[[1.]]])]
```

mygrad.broadcast_to

`mygrad.broadcast_to`(*a: ArrayLike*, *shape: Shape*, *, *constant: Optional[bool] = None*) → Tensor

Broadcast a tensor to a new shape.

This docstring was adapted from `numpy.broadcast_to`.

Parameters

a
[ArrayLike] The tensor to be broadcasted

shape: Tuple[int, ...]
The shape of the broadcasted tensor. This shape should be broadcast-compatible with the original shape.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Raises

ValueError
If the array is not compatible with the new shape according to Numpy's broadcasting rules.

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([1, 2, 3])
>>> mg.broadcast_to(x, (3,3))
Tensor([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]])
>>> mg.broadcast_to(x, (4,4))
Traceback (most recent call last) -> Tensor:
...
ValueError: operands could not be broadcast together with remapped
shapes [original->remapped]: (3,) and requested shape (4,4)
```

mygrad.expand_dims

`mygrad.expand_dims`(*a: ArrayLike*, *axis: int*, *, *constant: Optional[bool] = None*) → Tensor

Expand the dimensions of a tensor by adding a new axis.

This docstring was adapted from `numpy.expand_dims`.

Parameters

a
[ArrayLike] The tensor to be expanded

axis
[int] The position of the new axis in the expanded array shape.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([1, 2])
>>> x.shape
(2,)
>>> y = mg.expand_dims(x, 1)
>>> y.shape
(2, 1)
>>> z = mg.expand_dims(y, 0)
>>> z.shape
(1, 2, 1)
```

mygrad.squeeze

`mygrad.squeeze`(*a: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Remove single-dimensional entries from the shape of a tensor.

This docstring was adapted from `numpy.squeeze`

Parameters

a
[ArrayLike] The tensor to be reshaped

axis
[Optional[int, Tuple[int, . . .]]] Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Raises

ValueError
If `axis` is not `None`, and an axis being squeezed is not of length 1

Examples

```
>>> import mygrad as mg
>>> x = mg.Tensor([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> mg.squeeze(x).shape
(3,)
>>> mg.squeeze(x, axis=0).shape
(3, 1)
>>> mg.squeeze(x, axis=1).shape
Traceback (most recent call last) -> Tensor:
...
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> mg.squeeze(x, axis=2).shape
(1, 3)
```

3.8.4 Joining tensors

concatenate([axis, out, constant])	Join a sequence of tensors along an existing axis.
stack([axis, out, constant])	Join a sequence of tensors along a new axis.

mygrad.concatenate

mygrad.**concatenate**(*(t1, t2, ...)*, *axis=0*, *out=None*, ***, *constant=None*)

Join a sequence of tensors along an existing axis.

This docstring was adapted from that of numpy.concatenate [1]

Parameters

tensors
[Sequence[ArrayLike]] The tensors must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis
[Optional[int]] The axis along which the tensors will be joined. If axis is `None`, tensors are flattened before use. Default is 0.

out
[Optional[Union[ndarray, Tensor]]] If provided, the destination to place the result. The shape must be correct, matching that of what concatenate would have returned if no out argument were specified.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] If provided, the destination array will have this dtype. Cannot be provided together with `out`.

Requires numpy 1.20 or higher.

Returns

res
[Tensor] The concatenated tensor.

See also:

stack
Stack a sequence of tensors along a new axis.

`hstack`
Stack tensors in sequence horizontally (column wise).

`vstack`
Stack tensors in sequence vertically (row wise).

`dstack`
Stack tensors in sequence depth wise (along third dimension).

References

[1]

Examples

```
>>> import mygrad as mg
>>> a = mg.tensor([[1, 2], [3, 4]])
>>> b = mg.tensor([[5, 6]])
>>> mg.concatenate((a, b), axis=0)
Tensor([[1, 2],
       [3, 4],
       [5, 6]])
>>> mg.concatenate((a, b.T), axis=1)
Tensor([[1, 2, 5],
       [3, 4, 6]])
>>> mg.concatenate((a, b), axis=None)
Tensor([1, 2, 3, 4, 5, 6])
```

mygrad.stack

`mygrad.stack`((*t1*, *t2*, *...*), *axis=0*, *out=None*, ***, *constant=None*)

Join a sequence of tensors along a new axis.

This docstring was adapted from that of numpy.stack [1]

Parameters

tensors
[Sequence[ArrayLike]] Each tensor must have the same shape.

axis
[Optional[int]] The axis in the result tensor along which the input tensors are stacked.

out
[Optional[Union[ndarray, Tensor]]] If provided, the destination to place the result. The shape must be correct, matching that of what concatenate would have returned if no out argument were specified.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

res
[Tensor] The stacked tensor has one more dimension than the input arrays.

See also:

`concatenate`
Join a sequence of tensors along an existing axis.

`hstack`
Stack tensors in sequence horizontally (column wise).

`vstack`
Stack tensors in sequence vertically (row wise).

`dstack`
Stack tensors in sequence depth wise (along third dimension).

References

[1]

Examples

```
>>> import mygrad as mg
>>> a = mg.tensor([1, 2, 3])
>>> b = mg.tensor([-1, -2, -3])
>>> mg.stack((a, b))
Tensor([[ 1,  2,  3],
        [-1, -2, -3]])
```

```
>>> mg.stack((a, b), axis=-1)
Tensor([[1, -1],
        [2, -2],
        [3, -3]])
```

3.8.5 Tiling tensors

repeat(a, repeats[, axis, constant])	Repeat elements of a tensor.

mygrad.repeat

`mygrad.`**`repeat`**(*a: ArrayLike*, *repeats: Union[int, Sequence[int]]*, *axis: Optional[int] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Repeat elements of a tensor.

This docstring was adapted from `numpy.repeat`

Parameters

a
[ArrayLike] Input tensor.

repeats
[Union[int, Sequence[int]]] The number of repetitions for each element. `repeats` is broadcasted to fit the shape of the given axis.

axis
[Optional[int]] The axis along which to repeat values. By default, use the flattened input array, and return a flat output tensor.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

——-

repeated_tensor
[Tensor] Output tensor which has the same shape as *a*, except along the given axis.

Examples

```
>>> import mygrad as mg
>>> mg.repeat(3, 4)
Tensor([3, 3, 3, 3])
>>> x = mg.Tensor([[1, 2], [3, 4]])
>>> mg.repeat(x, 2)
Tensor([1, 1, 2, 2, 3, 3, 4, 4])
>>> mg.repeat(x, 3, axis=1)
Tensor([[1, 1, 1, 2, 2, 2],
        [3, 3, 3, 4, 4, 4]])
>>> mg.repeat(x, [1, 2], axis=0)
Tensor([[1, 2],
        [3, 4],
        [3, 4]])
```

3.9 Linear algebra (`mygrad.linalg`)

3.9.1 Matrix and vector products

matmul(x1, x2[, out, dtype, constant])	Matrix product of two tensors:
multi_matmul(tensors, *[, constant])	Matrix product of two or more tensors calculated in the optimal ordering
einsum(subscripts, *operands)	Evaluates the Einstein summation convention on the operands.

mygrad.matmul

`class mygrad.matmul`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Matrix product of two tensors:

`matmul(x, y)` is equivalent to `x @ y`.

This documentation was adapted from `numpy.matmul`

The behavior depends on the arguments in the following way.

- If both arguments are 2-D they are multiplied like conventional matrices.
- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

Multiplication by a scalar is not allowed, use * instead. Note that multiplying a stack of matrices with a vector will result in a stack of vectors, but matmul will not recognize it as such.

`matmul` differs from `numpy.dot` in two important ways.

- Multiplication by scalars is not allowed.
- Stacks of matrices are broadcast together as if the matrices were elements.

Parameters

x1
[ArrayLike]

x2
[ArrayLike]

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

Returns

output
[mygrad.Tensor] Returns the matrix product of `x1` and *x2*`.

Raises

ValueError

If :

- The last dimension of `x1` is not the same size as the second-to-last dimension of `x2`.
- If scalar value is passed.

See also:

einsum
Einstein summation convention.

Notes

The matmul function implements the semantics of the @ operator introduced in Python 3.5 following PEP465.

Examples

For two 2D tensors, `matmul(a, b)` is the matrix product $\sum_j A_{ij} B_{jk} = F_{ik}$:

```
>>> import mygrad as mg
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> mg.matmul(a, b)
Tensor([[4, 1],
        [2, 2]])
```

For 2-D mixed with 1-D, the result is the matrix-vector product, $\sum_j A_{ij} B_j = F_i$:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [1, 2]
>>> mg.matmul(a, b)
Tensor([1, 2])
```

Broadcasting is conventional for stacks of arrays. Here `a` is treated like a stack of three 5x6 matrices, and the 6x4 matrix `b` is broadcast matrix-multiplied against each one. This produces a shape-(3, 5, 4) tensor as a result.

```
>>> a = mg.arange(3*5*6).reshape((3,5,6))
>>> b = mg.arange(6*4).reshape((6,4))
>>> mg.matmul(a,b).shape
(3, 5, 4)
```

Scalar multiplication raises an error.

```
>>> mg.matmul(a, 3)
Traceback (most recent call last):
...
ValueError: Scalar operands are not allowed, use '*' instead
```

Attributes

identity

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.multi_matmul

`mygrad.multi_matmul`(*tensors: ArrayLike*, *, *constant: Optional[bool] = None*) → Tensor

Matrix product of two or more tensors calculated in the optimal ordering

This documentation was adapted from `numpy.linalg.multi_dot`

Compute the matrix multiplication of two or more arrays in a single function call, while automatically selecting the fastest evaluation order. `multi_matmul` chains `matmul` and uses optimal parenthesization [1] [2]. Depending on the shapes of the matrices, this can speed up the multiplication a lot.

If the first argument is 1-D it is treated as a row vector.

If the last argument is 1-D it is treated as a column vector.

The other arguments must be 2-D or greater.

Think of *multi_dot* as an optimized version of:

```
def multi_dot(tensors): return functools.reduce(mg.matmul, tensors)
```

Parameters

tensors: Sequence[array_like]
The sequence of tensors to be matrix-multiplied.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor
Returns the matrix product of the tensors provided

Raises

ValueError
If `tensors` contains less than two array_like items.

ValueError
If `tensor` other than the first or last is less than two dimensional

See also:

`matmul`
matrix multiplication with two arguments.

Notes

The cost for a matrix multiplication can be calculated with the following function:

```
def cost(A, B):
    return A.shape[0] * A.shape[1] * B.shape[1]
```

Let's assume we have three matrices $A_{10x100}, B_{100x5}, C_{5x50}$.

The costs for the two different parenthesizations are as follows:

```
cost((AB)C) = 10*100*5 + 10*5*50   = 5000 + 2500   = 7500
cost(A(BC)) = 10*100*50 + 100*5*50 = 50000 + 25000 = 75000
```

References

[1], [2]

Examples

multi_matmul allows you to write:

```
>>> from mygrad.math.misc.funcs import matmul    >>> from mygrad import multi_
↪matmul,  Tensor
>>> import numpy as np
>>> # Prepare some random tensors
>>> A = Tensor(np.random.random((10000, 100)))
>>> B = Tensor(np.random.random((100, 1000)))
>>> C = Tensor(np.random.random((1000, 5)))
>>> D = Tensor(np.random.random((5, 333)))
>>> # the actual matrix multiplication
>>> multi_matmul([A, B, C, D]) # computes (A @ (B @ C)) @ D
```

instead of:

```
>>> matmul(matmul(matmul(A, B), C), D)
>>> # or
>>> A @ B @ C @ D
```

mygrad.einsum

mygrad.**einsum**(*subscripts*, **operands*)

Evaluates the Einstein summation convention on the operands. This implementation exactly mirrors that of numpy.einsum and supports back-propagation through all variety of tensor-products, sums, traces, and views that it can perform.

The following docstring was adapted from the documentation for numpy.einsum

Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations. The best way to understand this function is to try the examples below, which show how many common NumPy/MyGrad functions can be implemented as calls to einsum.

Back-propagation via `einsum` is optimized such that any tensor that occurs redundantly within the summation will only have its gradient computed once. This optimization accommodates all number and combination of redundancies that can be encountered.

E.g. back-propping through `einsum('...,...->', x, x)` will only incur a single computation/accumulation for `x.grad` rather than two. This permits users to leverage the efficiency of sum-reduction, where `(x ** 2).sum()` is sub-optimal, without being penalized during back-propagation.

Parameters

subscripts
[str] Specifies the subscripts for summation.

operands
[array_like] The tensors used in the summation.

optimize
[{False, True, 'greedy', 'optimal'}, optional (default=False)] Controls if intermediate optimization should occur; also enables the use of BLAS where possible. This can produce significant speedups for computations like matrix multiplication.

No optimization will occur if False and True will default to the 'greedy' algorithm. Also accepts an explicit contraction list from the `np.einsum_path` function. See `np.einsum_path` for more details.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

output
[mygrad.Tensor] The calculation based on the Einstein summation convention.

Notes

The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand. Repeated subscripts labels in one operand take the diagonal. For example, `einsum('ii', a)` is equivalent to `np.trace(a)` (however, the former supports back-propagation).

Whenever a label is repeated, it is summed, so `einsum('i, i', a, b)` is equivalent to `np.inner(a, b)`. If a label appears only once, it is not summed, so `einsum('i', a)` produces a view of `a` with no changes.

The order of labels in the output is by default alphabetical. This means that `np.einsum('ij', a)` doesn't affect a 2D tensor, while `einsum('ji', a)` takes its transpose.

The output can be controlled by specifying output subscript labels as well. This specifies the label order, and allows summing to be disallowed or forced when desired. The call `einsum('i->', a)` is like `np.sum(a, axis=-1)`, and `einsum('ii->i', a)` is like `np.diag(a)`. The difference is that *einsum* does not allow broadcasting by default.

To enable and control broadcasting, use an ellipsis. Default NumPy-style broadcasting is done by adding an ellipsis to the left of each term, like `einsum('...ii->...i', a)`. To take the trace along the first and last axes, you can do `einsum('i...i', a)`, or to do a matrix-matrix product with the left-most indices instead of rightmost, you can do `einsum('ij...,jk...->ik...', a, b)`.

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new tensor. Thus, taking the diagonal as `einsum('ii->i', a)` produces a view.

An alternative way to provide the subscripts and operands is as `einsum(op0, sublist0, op1, sublist1, ..., [sublistout])`. The examples below have corresponding *einsum* calls with the two parameter methods.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.arange(25).reshape(5,5)
>>> b = mg.arange(5)
>>> c = mg.arange(6).reshape(2,3)
```

Compute the trace of `a`, $\sum_i A_{ii} = f$:

```
>>> einsum('ii', a)
Tensor(60)
>>> einsum(a, [0, 0])
Tensor(60)
>>> np.trace(a.data)
array(60)
```

Return a view along the diagonal of `a`, $A_{ii} = F_i$:

```
>>> einsum('ii->i', a)
Tensor([ 0,  6, 12, 18, 24])
>>> einsum(a, [0,0], [0])
Tensor([ 0,  6, 12, 18, 24])
>>> np.diag(a.data)
array([ 0,  6, 12, 18, 24])
```

Compute the matrix-vector product of `a` with `b`, $\sum_j A_{ij} B_j = F_i$:

```
>>> einsum('ij,j', a, b)
Tensor([ 30,  80, 130, 180, 230])
>>> einsum(a, [0,1], b, [1])
Tensor([ 30,  80, 130, 180, 230])
>>> mg.matmul(a, b)
Tensor([ 30,  80, 130, 180, 230])
>>> einsum('...j,j', a, b)
Tensor([ 30,  80, 130, 180, 230])
```

Take the transpose of `c`, $C_{ji} = F_{ij}$:

```
>>> einsum('ji', c)
Tensor([[0, 3],
        [1, 4],
        [2, 5]])
>>> einsum(c, [1, 0])
Tensor([[0, 3],
        [1, 4],
```

(continues on next page)

(continued from previous page)

```
        [2, 5]])
>>> c.T
Tensor([[0, 3],
        [1, 4],
        [2, 5]])
```

Compute `3 * c`:

```
>>> einsum('..., ...', 3, c)
Tensor([[ 0,  3,  6],
        [ 9, 12, 15]])
>>> einsum(',ij', 3, c)
Tensor([[ 0,  3,  6],
        [ 9, 12, 15]])
>>> einsum(3, [Ellipsis], c, [Ellipsis])
Tensor([[ 0,  3,  6],
        [ 9, 12, 15]])
>>> 3 * c
Tensor([[ 0,  3,  6],
        [ 9, 12, 15]])
```

Compute the inner product of `b` with itself, $\sum_i B_i B_i = f$:

```
>>> einsum('i,i', b, b)
Tensor(30)
>>> einsum(b, [0], b, [0])
Tensor(30)
>>> np.inner(b.data, b.data)
30
```

Compute the outer product of `array([1, 2])` with `b`, $A_i B_j = F_{ij}$:

```
>>> einsum('i,j', np.arange(2)+1, b)
Tensor([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> einsum(np.arange(2)+1, [0], b, [1])
Tensor([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> np.outer(np.arange(2)+1, b)
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> einsum('i...->...', a)
Tensor([50, 55, 60, 65, 70])
>>> einsum(a, [0,Ellipsis], [Ellipsis])
Tensor([50, 55, 60, 65, 70])
>>> np.sum(a, axis=0)
array([50, 55, 60, 65, 70])
```

Compute the tensor product $\sum_{ij} A_{ijk} B_{jil} = F_{kl}$

```
>>> a = mg.arange(60.).reshape(3,4,5)
>>> b = mg.arange(24.).reshape(4,3,2)
>>> einsum('ijk,jil->kl', a, b)
```

(continues on next page)

(continued from previous page)

```
Tensor([[ 4400.,  4730.],
        [ 4532.,  4874.],
        [ 4664.,  5018.],
        [ 4796.,  5162.],
        [ 4928.,  5306.]])
>>> einsum(a, [0,1,2], b, [1,0,3], [2,3])
Tensor([[ 4400.,  4730.],
        [ 4532.,  4874.],
        [ 4664.,  5018.],
        [ 4796.,  5162.],
        [ 4928.,  5306.]])
>>> np.tensordot(a,b, axes=([1,0],[0,1]))
array([[ 4400.,  4730.],
        [ 4532.,  4874.],
        [ 4664.,  5018.],
        [ 4796.,  5162.],
        [ 4928.,  5306.]])
```

Matrix multiply `a.T` with `b.T`, $\sum_k A_{ki} B_{jk} = F_{ij}$

```
>>> a = mg.arange(6).reshape((3,2))
>>> b = mg.arange(12).reshape((4,3))
>>> einsum('ki,jk->ij', a, b)
Tensor([[10, 28, 46, 64],
        [13, 40, 67, 94]])
>>> einsum('ki,...k->i...', a, b)
Tensor([[10, 28, 46, 64],
        [13, 40, 67, 94]])
>>> einsum('k...,jk', a, b)
Tensor([[10, 28, 46, 64],
        [13, 40, 67, 94]])
```

Make an assignment to a view along the diagonal of `a`:

```
>>> a = mg.zeros((3, 3))
>>> einsum('ii->i', a).data[:] = 1
>>> a
Tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

3.9.2 Norms and other numbers

`linalg.norm`(x[, ord, axis, keepdims, ...])	Vector norm.

mygrad.linalg.norm

`mygrad.linalg.norm`(*x: ArrayLike*, *ord: Optional[Union[int, float]] = None*, *axis: Optional[Union[int, Tuple[int]]] = None*, *keepdims: bool = False*, ***, *nan_to_num: bool = True*, *constant: Optional[bool] = None*) → Tensor

Vector norm.

This function is an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

In contrast to `numpy.linalg.norm`, matrix norms are not supported.

This docstring was adapted from that of `numpy.linalg.norm` [1].

Parameters

x
[ArrayLike] Input tensor. If *axis* is None, then *x* must be 1-D unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.

ord
[Optional[Union[int, float]]] Order of the norm (see table under `Notes`). inf means numpy's *inf* object. The default is None.

axis
[Optional[Union[int, Tuple[int]]]] If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. The default is None.

keepdims
[bool, optional (default=False)] If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original *x*.

nan_to_num
[bool, optional (default=True)] If *True* then gradients that would store nans due to the presence of zeros in *x* will instead store zeros in those places.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

Tensor
Norm(s) of the vector(s).

Notes

For values of `ord < 1`, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes.

The following norms can be calculated:

ord	norm for vectors
inf	max(abs(x))
-inf	min(abs(x))
0	sum(x != 0)
1	as below
-1	as below
2	as below
-2	as below
other	sum(abs(x)**ord)**(1./ord)

The Frobenius norm is given by [1]:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

The nuclear norm is the sum of the singular values.

Both the Frobenius and nuclear norm orders are only defined for matrices and raise a ValueError when `x.ndim != 2`.

References

[1], [2]

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor([[1.0, 2.0, 3.0],
...                [1.0, 0.0, 0.0]])
>>> l2_norms = mg.linalg.norm(x, axis=1, ord=2)
>>> l2_norms
Tensor([3.74165739, 1.        ])
```

The presence of the elementwise absolute values in the norm operation means that zero-valued entries in any of input vectors have an undefined derivative. When *nan_to_num=False* is specified these derivatives will be reported as *nan*, otherwise they will be made to be 0.0.

```
>>> l2_norms = mg.linalg.norm(x, axis=1, ord=2, nan_to_num=True)
>>> l2_norms.backward()
>>> x.grad
array([[0.26726124, 0.53452248, 0.80178373],
       [1.        ,        nan,        nan]])
```

This is rigorously true, but is often not the desired behavior in autodiff applications. Rather, it can be preferable to use *0.0* to fill these undefined derivatives. This is the default behavior, when *nan_to_num* is not specified.

```
>>> l2_norms = mg.linalg.norm(x, axis=1, ord=2, nan_to_num=False)  # default␣
↪setting: `nan_to_num=False`
>>> l2_norms.backward()
>>> x.grad
array([[0.26726124, 0.53452248, 0.80178373],
      [1.        ,         0.,         0.]])
```

L1 norms along each of the three columns:

```
>>> mg.linalg.norm(x, axis=0, ord=1)
Tensor([2., 2., 3.])
```

3.10 Mathematical functions (`mygrad.math`)

3.10.1 Trigonometric functions

sin(x[, out, where, dtype, constant])	Trigonometric sine, element-wise.
cos(x[, out, where, dtype, constant])	Trigonometric cosine, element-wise.
tan(x[, out, where, dtype, constant])	Trigonometric tangent, element-wise.
arcsin(x[, out, where, dtype, constant])	Inverse sine, element-wise.
arccos(x[, out, where, dtype, constant])	Inverse cosine, element-wise.
arctan(x[, out, where, dtype, constant])	Inverse tangent, element-wise.
arctan2(x1, x2[, out, where, dtype, constant])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

mygrad.sin

class `mygrad.sin`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Trigonometric sine, element-wise.

This docstring was adapted from that of numpy.sin [1]

Parameters

x
[ArrayLike] Angle, in radians (2π rad equals 360 degrees).

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True,

the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The sine of each element of x.

See also:

arcsin, *sinh*, *cos*

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.sin(mg.pi/2.)
Tensor(1.0)
```

Print sines of an array of angles given in degrees:

```
>>> mg.sin(mg.tensor((0., 30., 45., 60., 90.)) * mg.pi / 180. )
Tensor([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.cos

class `mygrad.cos`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Trigonometric cosine, element-wise.

This docstring was adapted from that of numpy.cos [1]

Parameters

x

[ArrayLike] Input array in radians.

out
: [Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
: [Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

  Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

  Integer-type tensors must be constant.

where
: [Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
: [Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
: [Tensor] The corresponding cosine values.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

[1]

Examples

```
>>> import mygrad as mg
>>> mg.cos([0, mg.pi/2, mg.pi])
Tensor([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.tan

class `mygrad.`**`tan`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Trigonometric tangent, element-wise.

This docstring was adapted from that of numpy.tan [1]

Parameters

x
[ArrayLike] Input array.

out
: [Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
: [Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

  Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

  Integer-type tensors must be constant.

where
: [Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
: [Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
: [Tensor] The corresponding tangent values.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

[1]

Examples

```
>>> import mygrad as mg
>>> from math import pi
>>> mg.tan([-pi, pi / 2, pi])
Tensor([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arcsin

class `mygrad.`**`arcsin`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse sine, element-wise.

This docstring was adapted from that of numpy.arcsin [1]

Parameters

x

[ArrayLike] *y*-coordinate on the unit circle.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

angle
[Tensor] The inverse sine of each element in *x*, in radians and in the closed interval `[-pi/2, pi/2]`.

See also:

`sin`, `cos`, `arccos`, `tan`, `arctan`, `arctan2`

Notes

arcsin is a multivalued function: for each x there are infinitely many numbers z such that $sin(z) = x$. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arcsin* is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or sin^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

[1]

Examples

```
>>> import mygrad as mg
>>> mg.arcsin(1)     # pi/2
Tensor(1.5707963267948966)
>>> mg.arcsin(-1)    # -pi/2
Tensor(-1.5707963267948966)
>>> mg.arcsin(0)
Tensor(0.0)
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args, **kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arccos

class `mygrad.arccos`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse cosine, element-wise.

This docstring was adapted from that of numpy.arccos [1]

Parameters

x
[ArrayLike] *x*-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

angle
[Tensor] The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi].

See also:

`cos`, `arctan`, `arcsin`

Notes

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z) = x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, *arccos* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytic function that has branch cuts *[-inf, -1]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse *cos* is also known as *acos* or cos^-1.

References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

[1]

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> import mygrad as mg
>>> mg.arccos([1, -1])
Tensor([ 0.        ,  3.14159265])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arctan

class `mygrad.`**`arctan`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse tangent, element-wise.

This docstring was adapted from that of numpy.arctan [1]

Parameters

x
[ArrayLike]

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

out
[Tensor]

See also:

`arctan2`
The "four quadrant" arctan of the angle formed by (x, y) and the positive x-axis.

Notes

arctan is a multi-valued function: for each x there are infinitely many numbers z such that $\tan(z) = x$. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arctan* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctan* is a complex analytic function that has [*1j, infj*] and [-*1j, -infj*] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or tan^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

[1]

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> import mygrad as mg
>>> mg.arctan([0, 1])
Tensor([ 0.        ,  0.78539816])
```

```
>>> mg.pi / 4
0.78539816339744828
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arctan2

class `mygrad.`**`arctan2`**(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

This docstring was adapted from that of numpy.arctan [1]

Parameters

x1
[ArrayLike] y-coordinates.

x2
[ArrayLike] `x`-coordinates.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

angle
[Tensor] Tensor of angles in radians, in the range `[-pi, pi]`.

See also:

`arctan`, `tan`

Notes

arctan2 is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard: [2]

x1	*x2*	*arctan2(x1,x2)*
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/-inf	+0 / +pi
< 0	+/-inf	-0 / -pi
+/-inf	+inf	+/- (pi/4)
+/-inf	-inf	+/- (3*pi/4)

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

References

[1], [2]

Examples

Consider four points in different quadrants:

```
>>> import mygrad as mg
>>> x = mg.tensor([-1.0, +1.0, +1.0, -1.0])
>>> y = mg.tensor([-1.0, -1.0, +1.0, +1.0])
>>> mg.arctan2(y, x) * 180 / mg.pi
Tensor([-135.,  -45.,   45.,  135.])
```

Note the order of the parameters. *arctan2* is defined also when *x2* = 0 and at several other special points, obtaining values in the range `[-pi, pi]`:

```
>>> mg.arctan2([1., -1.], [0., 0.])
Tenor([ 1.57079633, -1.57079633])
>>> mg.arctan2([0., 0., mg.inf], [+0., -0., mg.inf])
Tenor([ 0.        ,  3.14159265,  0.78539816])
```

Attributes

identity

signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.10.2 Hyperbolic functions

sinh(x[, out, where, dtype, constant])	Hyperbolic sine, element-wise.
cosh(x[, out, where, dtype, constant])	Hyperbolic cosine, element-wise.
tanh(x[, out, where, dtype, constant])	Hyperbolic tangent, element-wise.
arcsinh(x[, out, where, dtype, constant])	Inverse hyperbolic sine, element-wise.
arccosh(x[, out, where, dtype, constant])	Inverse hyperbolic cosine, element-wise.
arctanh(x[, out, where, dtype, constant])	Inverse hyperbolic tangent, element-wise.

mygrad.sinh

`class mygrad.sinh`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, *, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Hyperbolic sine, element-wise.

This docstring was adapted from that of numpy.sinh [1]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`). Defaults to `False` for float-type data. Defaults to `True` for integer-type data. Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original

value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The corresponding hyperbolic sine values.

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

[1]

Examples

```
>>> import mygrad as mg
>>> mg.sinh(0)
Tensor(0.0)
```

```
>>> # Example of providing the optional output tensor
>>> out1 = mg.tensor([0], dtype='d')
>>> out2 = mg.sinh([0.1], out=out1)
>>> out2
Tensor([0.10016675])
>>> out2 is out1
True
```

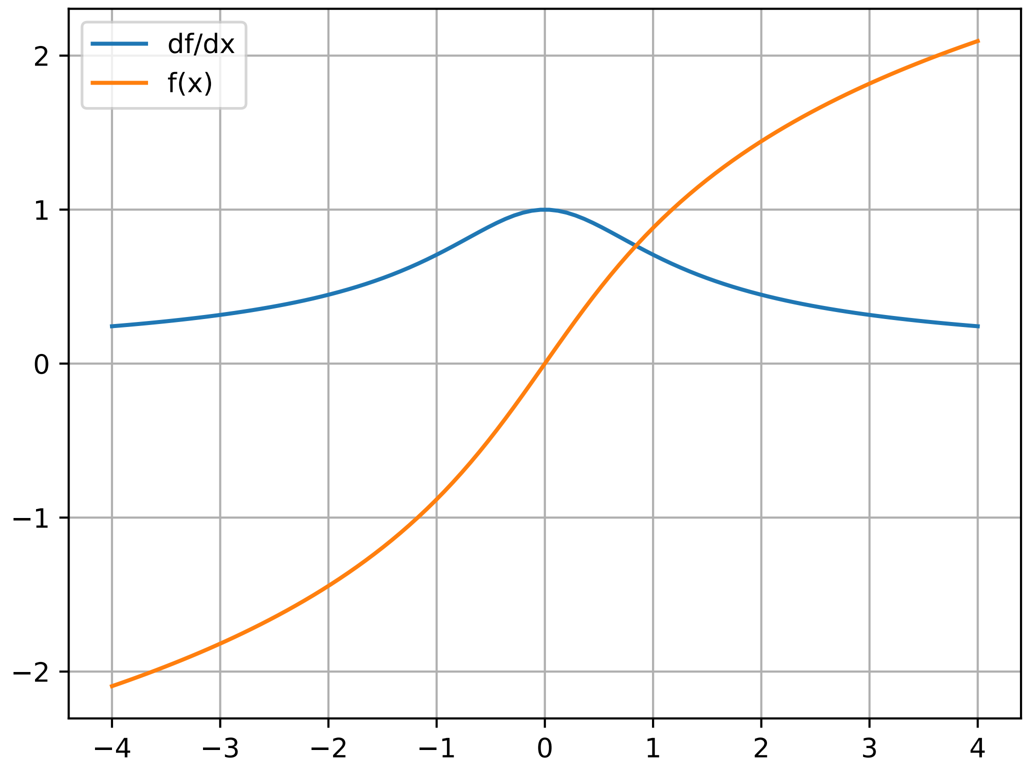
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


df/dx
f(x)
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3
4

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.cosh

class `mygrad.cosh`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, *, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Hyperbolic cosine, element-wise.

This docstring was adapted from that of numpy.cosh [1]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

out
[Tensor] Output tensor of same shape as *x*.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.linspace(-2, 2, 10)
>>> mg.cosh(x); y
Tensor([3.76219569, 2.47439497, 1.68346238, 1.23057558, 1.02479314,
    1.02479314, 1.23057558, 1.68346238, 2.47439497, 3.76219569])
```

```
>>> y.backward()  # compute d(cosh)/dx
>>> x.grad
array([-3.62686041, -2.26332289, -1.35426939, -0.71715846, -0.22405573,
    0.22405573,  0.71715846,  1.35426939,  2.26332289,  3.62686041])
```

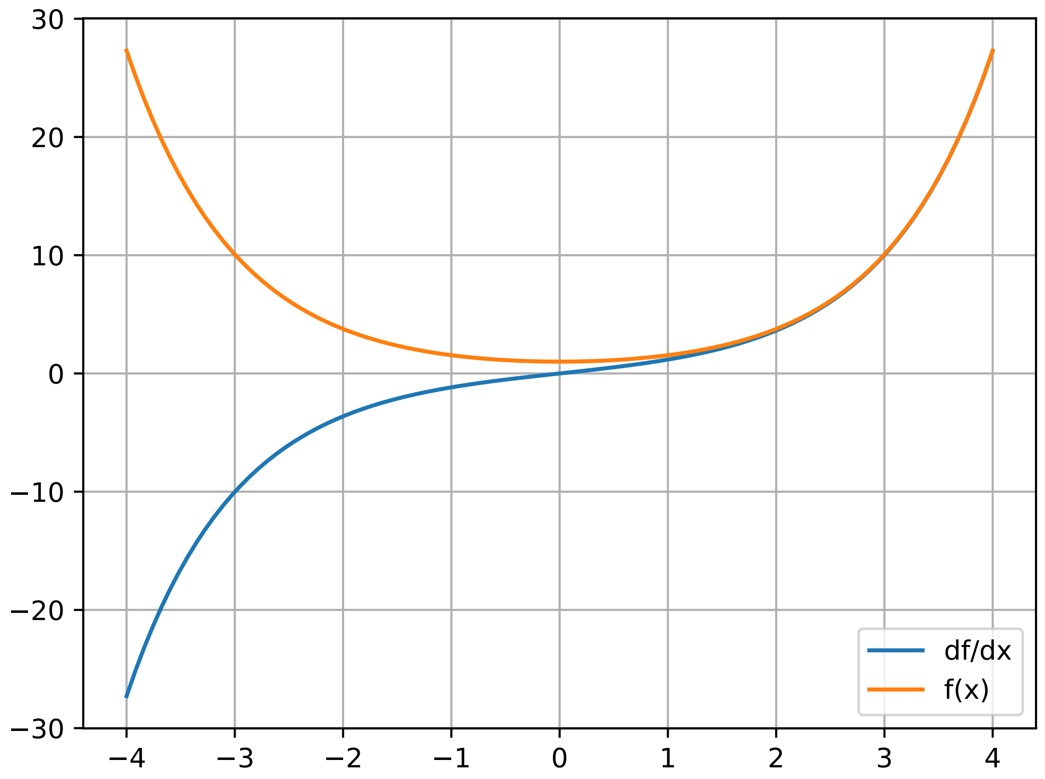
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


30
20
10
0
−10
−20
−30
−4
−3
−2
−1
0
1
2
3
4
df/dx
f(x)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.tanh

`class mygrad.tanh`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Hyperbolic tangent, element-wise.

This docstring was adapted from that of numpy.tanh [3]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The corresponding hyperbolic tangent values.

References

[1], [2], [3]

Examples

```
>>> import mygrad as mg
>>> x = mg.linspace(-2, 2, 10)
>>> y = mg.tanh(x); y
Tensor([-0.96402758, -0.9146975 , -0.8044548 , -0.58278295, -0.21863508,
     0.21863508,  0.58278295,  0.8044548 ,  0.9146975 ,  0.96402758])
```

```
>>> y.backward()  # compute d(tanh)/dx
>>> x.grad
array([0.07065082, 0.16332849, 0.35285247, 0.66036404, 0.9521987 ,
       0.9521987 , 0.66036404, 0.35285247, 0.16332849, 0.07065082])
```

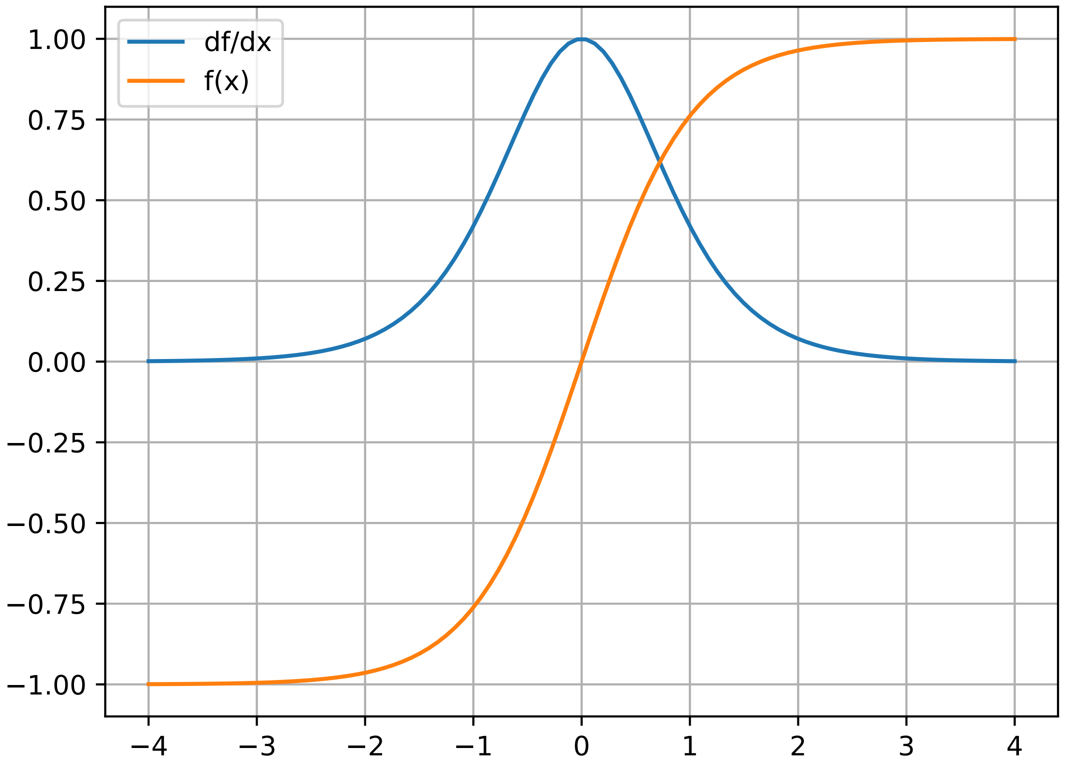
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


df/dx
f(x)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−4
−3
−2
−1
0
1
2
3
4

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arcsinh

class `mygrad.`**`arcsinh`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse hyperbolic sine, element-wise.

This docstring was adapted from that of numpy.arcsinh [3]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`). Defaults to `False` for float-type data. Defaults to `True` for integer-type data. Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True,

the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value.

Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

out
[Tensor] Tensor of the same shape as *x*.

Notes

arcsinh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *sinh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arcsinh* always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the *invalid* floating point error flag.

The inverse hyperbolic sine is also known as *asinh* or `sinh^-1`.

References

[1], [2], [3]

Examples

```
>>> import mygrad as mg
>>> mg.arcsinh([mg.e, 10.0])
Tensor([ 1.72538256,  2.99822295])
```

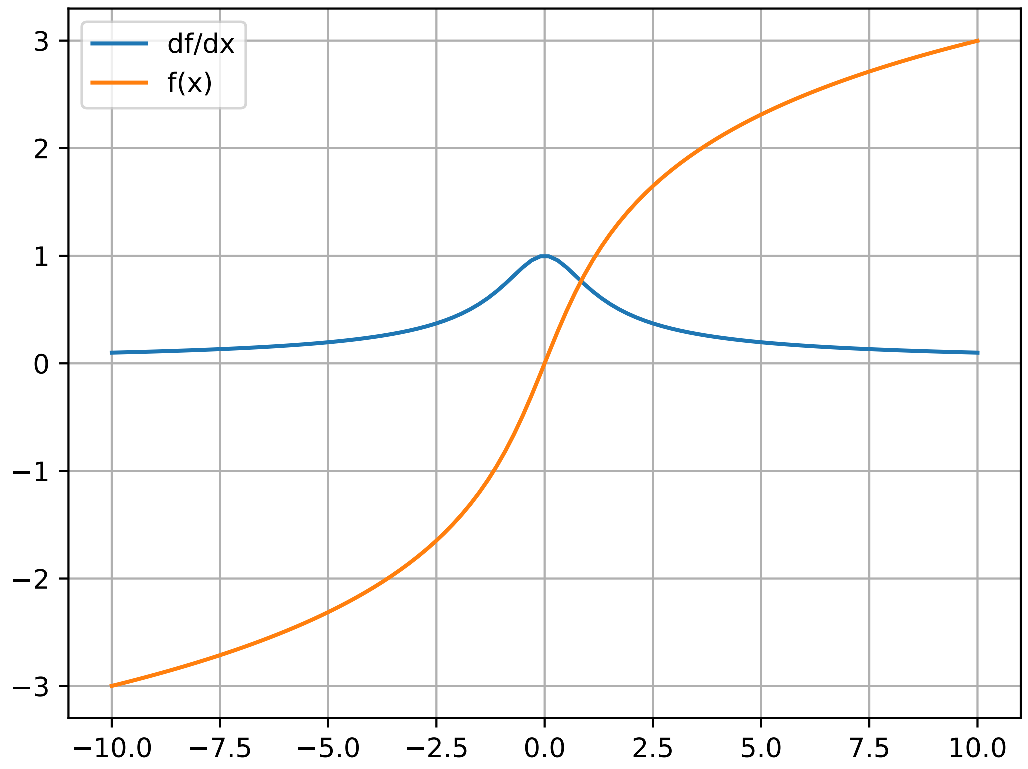
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


df/dx
f(x)
3
2
1
0
−1
−2
−3
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arccosh

class mygrad.**arccosh**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, *, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse hyperbolic cosine, element-wise.

This docstring was adapted from that of numpy.arccosh [3]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`). Defaults to `False` for float-type data. Defaults to `True` for integer-type data. Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original

value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

arccosh
[Tensor] Tensor of the same shape as *x*.

See also:

`cosh`, *`arcsinh`*, *`sinh`*, *`arctanh`*, *`tanh`*

Notes

arccosh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cosh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]* and the real part in `[0, inf]`.

For real-valued input data types, *arccosh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

References

[1], [2], [3]

Examples

```
>>> import mygrad as mg
>>> mg.arccosh([mg.e, 10.0])
Tensor([ 1.65745445,  2.99322285])
>>> mg.arccosh(1)
Tensor(0.0)
```

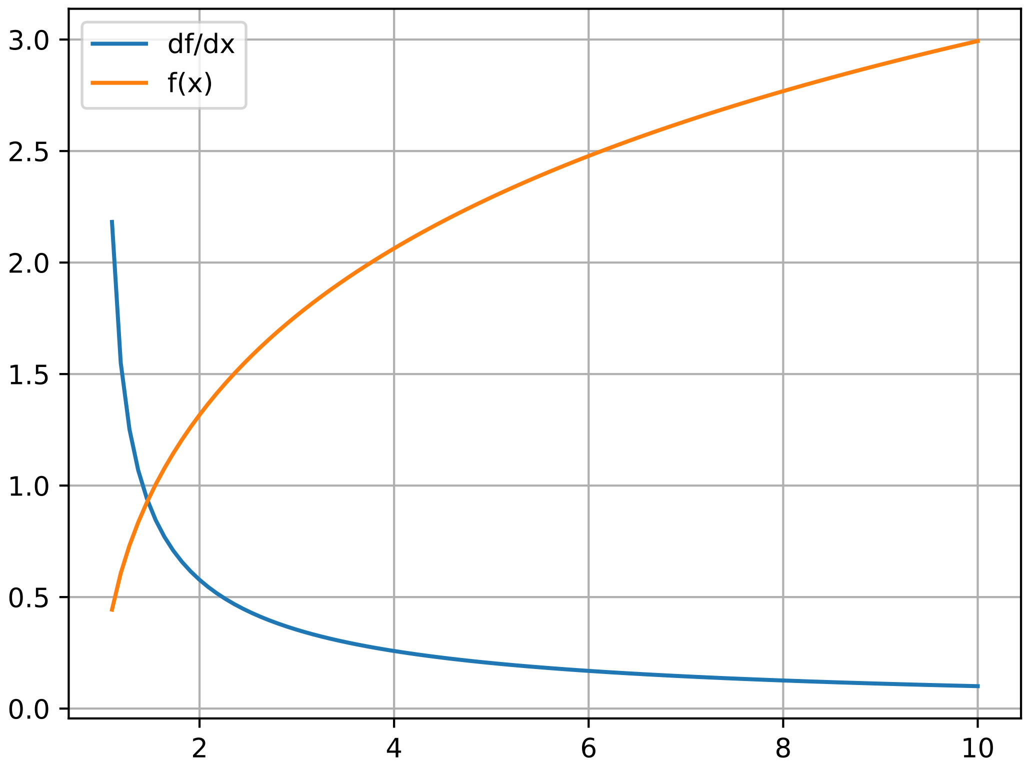
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


df/dx
f(x)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.arctanh

class `mygrad.`**`arctanh`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Inverse hyperbolic tangent, element-wise.

This docstring was adapted from that of numpy.arctanh [3]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`). Defaults to `False` for float-type data. Defaults to `True` for integer-type data. Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original

value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

out
[Tensor] Tensor of the same shape as *x*.

Notes

arctanh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *tanh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arctanh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

The inverse hyperbolic tangent is also known as *atanh* or `tanh^-1`.

References

[1], [2], [3]

Examples

```
>>> import mygrad as mg
>>> mg.arctanh([0, -0.5])
Tensor([ 0.        , -0.54930614])
```

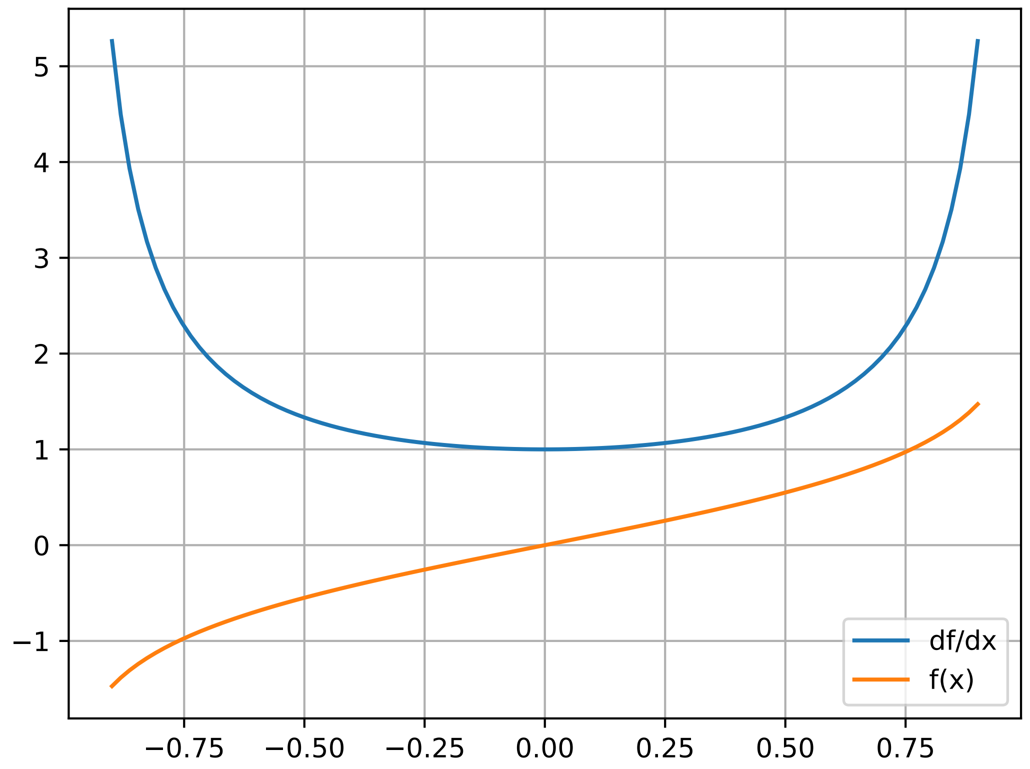
Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)


df/dx
f(x)
5
4
3
2
1
0
−1
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.10.3 Sums, products, differences

`prod`(a[, axis, keepdims, constant])	Return the product of array elements over given axes.
`sum`(x[, axis, keepdims, constant])	Sum of tensor elements over a given axis.
`cumprod`(a[, axis, constant])	Return the cumulative product of elements along a given axis.
`cumsum`(a[, axis, constant])	Return the cumulative sum of the elements along a given axis.
`mean`(x[, axis, keepdims, constant])	Mean of tensor elements over a given axis.
`var`(x[, axis, ddof, keepdims, constant])	Compute the variance along the specified axis.
`std`(x[, axis, ddof, keepdims, constant])	Compute the standard deviation along the specified axis.
`amax`(x[, axis, keepdims, constant])	Return the maximum of a tensor or maximum along its axes.
`amin`(x[, axis, keepdims, constant])	Return the minimum of a tensor or minimum along its axes.
`max`(x[, axis, keepdims, constant])	Return the maximum of a tensor or maximum along its axes.
`min`(x[, axis, keepdims, constant])	Return the minimum of a tensor or minimum along its axes.

mygrad.prod

`mygrad.prod`(*a: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Return the product of array elements over given axes.

Parameters

a
[ArrayLike] Input data.

axis
[Optional[int, Tuple[int, ...]]] Axis or axes along which to operate. By default, flattened input is used.

keepdims
[bool, optional (default=False)] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

product_along_axis
[mygrad.Tensor] A tensor shaped as *a* but with the specified axis removed.

Notes

The product of an empty tensor is the neutral element 1:

```
>>> import mygrad
>>> mygrad.prod([])
Tensor(1.0)
```

Examples

By default, calculate the product of all elements:

```
>>> import mygrad as mg
>>> mg.prod([1.,2.])
Tensor(2.0)
```

Even when the input array is two-dimensional:

```
>>> mg.prod([[1.,2.],
...          [3.,4.]])
Tensor(24.0)
```

But we can also specify the axis over which to multiply:

```
>>> mg.prod([[1.,2.],
...          [3.,4.]], axis=1)
Tensor([  2.,  12.])
```

mygrad.sum

`mygrad.sum`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Sum of tensor elements over a given axis.

Parameters

x
[ArrayLike]

axis
[Optional[int, Tuple[ints, . . .]]] Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input tensor. If axis is negative it counts from the last to the first axis. If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

sum_along_axis
[mygrad.Tensor] A Tensor with the same shape as *self*, with the specified axis/axes removed. If *self* is a 0-d tensor, or if *axis* is None, a 0-dim Tensor is returned.

See also:

`mygrad.Tensor.sum`
Equivalent method.

`cumsum`
Cumulative sum of array elements.

`mean`, **`average`**

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

The sum of an empty tensor is the neutral element 0:

```
>>> mygrad.sum([])
Tensor(0.0)
```

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> mg.sum([0.5, 1.5])
Tensor(2.0)
>>> mg.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
Tensor(1)
>>> mg.sum([[0, 1], [0, 5]])
Tensor(6)
>>> mg.sum([[0, 1], [0, 5]], axis=0)
Tensor([0, 6])
>>> mg.sum([[0, 1], [0, 5]], axis=1)
Tensor([1, 5])
```

If the accumulator is too small, overflow occurs:

```
>>> mg.ones(128, dtype=mg.int8).sum(dtype=np.int8)
Tensor(-128)
```

mygrad.cumprod

`mygrad.`**`cumprod`**(*a: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return the cumulative product of elements along a given axis.

This docstring was adapted from the official numpy documentation

Parameters

a
[ArrayLike] Input array.

axis
[Optional[int]] Axis along which the cumulative product is computed. By default the input is flattened.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> from mygrad import cumprod, Tensor
>>> a = Tensor([[1, 2, 3],
...             [4, 5, 6]])
```

```
>>> cumprod(a)
Tensor([  1   2   6  24 120 720])
```

The cumulative product for each column (i.e., over the rows) of *a*:

```
>>> cumprod(a, axis=0)
Tensor([[ 1,  2,  3],
       [ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns) of *a*:

```
>>> cumprod(a, axis=1)
Tensor([[  1,   2,   6],
        [  4,  20, 120]])
```

mygrad.cumsum

mygrad.**cumsum**(*a: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Return the cumulative sum of the elements along a given axis.

This docstring was adapted from the official numpy documentation

Parameters

a
[ArrayLike] Input array.

axis
[int, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

Examples

```
>>> from mygrad import cumsum, Tensor
>>> a = Tensor([[1, 2, 3],
...             [4, 5, 6]])
>>> cumsum(a)
Tensor([ 1,  3,  6, 10, 15, 21])
```

```
>>> cumsum(a, axis=0)      # sum over rows for each of the 3 columns
Tensor([[1, 2, 3],
        [5, 7, 9]])
>>> cumsum(a, axis=1)      # sum over columns for each of the 2 rows
Tensor([[ 1,  3,  6],
        [ 4,  9, 15]])
```

mygrad.mean

`mygrad.mean`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Mean of tensor elements over a given axis.

Parameters

x
[ArrayLike]

axis
[Optional[int, Tuple[ints, ...]] Axis or axes along which a mean is performed. The default, axis=None, will mean all of the elements of the input tensor. If axis is negative it counts from the last to the first axis.

If axis is a tuple of ints, a mean is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mean_along_axis
[Tensor] A Tensor with the same shape as *self*, with the specified axis/axes removed. If *self* is a 0-d tensor, or if *axis* is None, a 0-dim Tensor is returned.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.Tensor([[1, 2],
...                [3, 4]])
>>> mg.mean(a)
Tensor(2.5)
>>> mg.mean(a, axis=0)
Tensor([ 2.,  3.])
>>> mg.mean(a, axis=1)
Tensor([ 1.5,  3.5])
```

In single precision, *mean* can be inaccurate:

```
>>> a = mg.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mg.mean(a)
Tensor(0.54999924)
```

Computing the mean in float64 is more accurate:

```
>>> mg.mean(a, dtype=np.float64)
Tensor(0.55000000074505806)
```

mygrad.var

`mygrad.var`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *ddof: int = 0*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

x
[ArrayLike] Array containing numbers whose variance is desired.

axis
[Optional[int, Tuple[int, ...]]] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

ddof
[int, optional (default=0)] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims
[bool, optional (default=False)] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array..

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

——-

variance
[mygrad.Tensor]

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.Tensor([[1, 2],
...                [3, 4]])
>>> mg.var(a)
Tensor(1.25)
>>> mg.var(a, axis=0)
Tensor([ 1.,  1.])
>>> mg.var(a, axis=1)
Tensor([ 0.25,  0.25])
```

In single precision, `var()` can be inaccurate:

```
>>> a = mg.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mg.var(a)
Tensor(0.20250003)
```

Computing the variance in float64 is more accurate:

```
>>> mg.var(a, dtype=np.float64)
Tensor(0.20249999932944759)
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
Tensor(0.2025)
```

mygrad.std

`mygrad.std`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *ddof: int = 0*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Compute the standard deviation along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

x
[ArrayLike] Array containing numbers whose standard deviation is desired.

axis
[Optional[int, Tuple[int, ...]]] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

ddof
[int, optional (default=0)] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims
[bool, optional (default=False)] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

std
[mygrad.Tensor]

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.Tensor([[1, 2],
...                [3, 4]])
>>> mg.std(a)
Tensor(1.1180339887498949)
>>> mg.std(a, axis=0)
Tensor([ 1.,  1.])
>>> mg.std(a, axis=1)
Tensor([ 0.5,  0.5])
```

In single precision, `var()` can be inaccurate:

```
>>> a = mg.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mg.std(a)
Tensor(0.45000005)
```

Computing the variance in float64 is more accurate:

```
>>> mg.std(a, dtype=np.float64)
Tensor(0.44999999925494177)
```

mygrad.amax

`mygrad.amax`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Return the maximum of a tensor or maximum along its axes.

Parameters

x
[ArrayLike]

axis
[Optional[int, Tuple[int, ...]]] Axis or axes along which to operate. By default, flattened input is used.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

max
[mygrad.Tensor] Maximum of *a*. If *axis* is None, the result is a 0-D tensor.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.arange(4).reshape((2,2))
>>> a
Tensor([[0, 1],
        [2, 3]])
>>> mg.amax(a)           # Maximum of the flattened array
Tensor(3)
>>> mg.amax(a, axis=0)   # Maxima along the first axis
Tensor([2, 3])
>>> mg.amax(a, axis=1)   # Maxima along the second axis
Tensor([1, 3])
>>> b = mg.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> mg.amax(b)
Tensor(nan)
```

mygrad.amin

`mygrad.amin`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Return the minimum of a tensor or minimum along its axes.

Parameters

axis
[Optional[int, Tuple[int, …]]] Axis or axes along which to operate. By default, flattened input is used.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

min
[mygrad.Tensor] Minimum of *a*. If *axis* is None, the result is a 0-D tensor.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.arange(4).reshape((2,2))
>>> a
Tensor([[0, 1],
        [2, 3]])
>>> mg.amin(a)           # Minimum of the flattened array
Tensor(0)
>>> mg.amin(a, axis=0)   # Minima along the first axis
Tensor([0, 1])
>>> mg.amin(a, axis=1)   # Minima along the second axis
Tensor([0, 2])
>>> b = mg.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> mg.amin(b)
Tensor(nan)
```

mygrad.max

`mygrad.`**`max`**(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Return the maximum of a tensor or maximum along its axes.

Parameters

x
[ArrayLike]

axis
[Optional[int, Tuple[int, ...]]] Axis or axes along which to operate. By default, flattened input is used.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

max
[mygrad.Tensor] Maximum of *a*. If *axis* is None, the result is a 0-D tensor.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.arange(4).reshape((2,2))
>>> a
Tensor([[0, 1],
        [2, 3]])
>>> mg.amax(a)           # Maximum of the flattened array
Tensor(3)
>>> mg.amax(a, axis=0)   # Maxima along the first axis
Tensor([2, 3])
>>> mg.amax(a, axis=1)   # Maxima along the second axis
Tensor([1, 3])
>>> b = mg.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> mg.amax(b)
Tensor(nan)
```

mygrad.min

`mygrad.min`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = None*, *keepdims: bool = False*, ***, *constant: Optional[bool] = None*) → Tensor

Return the minimum of a tensor or minimum along its axes.

Parameters

axis
[Optional[int, Tuple[int, . . .]]] Axis or axes along which to operate. By default, flattened input is used.

keepdims
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

min
[mygrad.Tensor] Minimum of *a*. If *axis* is None, the result is a 0-D tensor.

Examples

```
>>> import mygrad as mg
>>> import numpy as np
>>> a = mg.arange(4).reshape((2,2))
>>> a
Tensor([[0, 1],
        [2, 3]])
>>> mg.amin(a)           # Minimum of the flattened array
Tensor(0)
>>> mg.amin(a, axis=0)   # Minima along the first axis
Tensor([0, 1])
>>> mg.amin(a, axis=1)   # Minima along the second axis
Tensor([0, 2])
>>> b = mg.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> mg.amin(b)
Tensor(nan)
```

3.10.4 Exponents and logarithms

exp(x1[, out, where, dtype, constant])	Calculate the exponential of all elements in the input tensor.
expm1(x1[, out, where, dtype, constant])	Calculate `exp(x) - 1` for all elements in the tensor.
exp2(x1[, out, where, dtype, constant])	Calculate *2**p* for all *p* in the input tensor.
log(x1[, out, where, dtype, constant])	Natural logarithm, element-wise.
log10(x1[, out, where, dtype, constant])	Return the base 10 logarithm of the input tensor, element-wise.
log2(x1[, out, where, dtype, constant])	Base-2 logarithm applied elementwise to the tensor.
log1p(x1[, out, where, dtype, constant])	Return the natural logarithm of one plus the input tensor, element-wise.
logaddexp(x1, x2[, out, where, dtype, constant])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x1, x2[, out, where, dtype, constant])	Logarithm of the sum of exponentiations of the inputs in base-2.

mygrad.exp

class `mygrad.exp`(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Calculate the exponential of all elements in the input tensor.

This docstring was adapted from that of numpy.exp [1]

Parameters

x1
[ArrayLike] Input values.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

exp
[Tensor] `f(x1)` computed element-wise

See also:

expm1
Calculate `exp(x) - 1` for all elements in the tensor.

exp2
Calculate `2**x` for all elements in the tensor.

Notes

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor(1.)
>>> f = mg.exp(x); f  # f(1.)
Tensor(2.71828183)
```

Evaluate df/dx at `x = 1`.

```
>>> f.backward()
>>> x.grad
>>> x.grad  # df/dx @ x=1
array(2.71828183)
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.expm1

class mygrad.**expm1**(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Calculate `exp(x) - 1` for all elements in the tensor.

This docstring was adapted from that of numpy.expm1 [1]

Parameters

x1
[ArrayLike] Input values.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

expm1
[Tensor] `f(x1)` computed element-wise

See also:

log1p
`log(1 + x)`, the inverse of expm1.

Notes

This function provides greater precision than `exp(x) - 1` for small values of `x`.

References

[1]

Examples

The true value of `exp(1e-10) - 1` is `1.00000000005e-10` to about 32 significant digits. This example shows the superiority of expm1 in this case.

```
>>> import mygrad as mg
>>> mg.expm1(1e-10)
Tensor(1.00000000005e-10)
>>> mg.exp(1e-10) - 1
Tensor(1.000000082740371e-10)
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.exp2

`class mygrad.exp2`(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Calculate *2**p* for all *p* in the input tensor.

This docstring was adapted from that of numpy.exp2 [1]

Parameters

x1
[ArrayLike] Input values.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

exp2
[Tensor] `f(x1)` computed element-wise

See also:

power

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.exp2([2., 3.])
Tensor([ 4.,  8.])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args, **kwargs*)

Methods

`__init__`(*args, **kwargs)

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.log

class `mygrad.log`(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Natural logarithm, element-wise.

The natural logarithm `log` is the inverse of the exponential function, so that `log(exp(x)) = x`. The natural logarithm is logarithm in base `e`.

This docstring was adapted from that of numpy.log [1]

Parameters

x1
[ArrayLike] Input value.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

log
[Tensor] `f(x1)` computed element-wise

See also:

`log10`, `log2`, `log1p`

Notes

Logarithm is a multivalued function: for each x there is an infinite number of z such that $exp(z) = x$. The convention is to return the z whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.log([1, mg.e, mg.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.log10

class `mygrad.`**`log10`**(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Return the base 10 logarithm of the input tensor, element-wise.

This docstring was adapted from that of numpy.log10 [1]

Parameters

x1
[ArrayLike] Input values.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

log10
[Tensor] `f(x1)` computed element-wise

Notes

For real-valued input data types, *log10* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.log10([1e-15, -3.])
Tensor([-15.,  nan])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.log2

class `mygrad.log2`(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Base-2 logarithm applied elementwise to the tensor.

This docstring was adapted from that of numpy.log2 [1]

Parameters

x1
[ArrayLike] Input values.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

log2
[Tensor] `f(x1)` computed element-wise

See also:

`log`, `log10`, `log1p`

Notes

For real-valued input data types, *log2* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor([0, 1, 2, 2**4])
>>> mg.log2(x)
Tensor([-Inf,   0.,   1.,   4.])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.log1p

class `mygrad.log1p`(*x1: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Return the natural logarithm of one plus the input tensor, element-wise.

Calculates `log(1 + x)`.

This docstring was adapted from that of numpy.log1p [1]

Parameters

x1

[ArrayLike] Input values.

out

[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant

[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

log1p
[Tensor] `f(x1)` computed element-wise

See also:

`expm1`
`exp(x) - 1`, the inverse of *log1p*.

Notes

For real-valued input, *log1p* is accurate also for *x* so small that *1 + x == 1* in floating-point accuracy.

For real-valued input data types, *log1p* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.log1p(1e-99)
1e-99
>>> mg.log(1 + 1e-99)
0.0
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.logaddexp

class mygrad.**logaddexp**(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Logarithm of the sum of exponentiations of the inputs.

Calculates `log(exp(x1) + exp(x2))`. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

This docstring was adapted from that of numpy.logaddexp [1]

Parameters

x1, x2
[ArrayLike] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

logaddexp
[Tensor] Logarithm of `exp(x1) + exp(x2)`.

See also:

`logaddexp2`
Logarithm of the sum of exponentiations of inputs in base 2.

References

[1]

Examples

```
>>> import mygrad as mg
>>> prob1 = mg.log(1e-50)
>>> prob2 = mg.log(2.5e-50)
>>> prob12 = mg.logaddexp(prob1, prob2)
>>> prob12
Tensor(-113.87649168120691)
>>> mg.exp(prob12)
Tensor(3.5000000000000057e-50)
```

Attributes

signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.logaddexp2

`class mygrad.logaddexp2`(*x1: ArrayLike, x2: ArrayLike, out: Optional[Union[Tensor, ndarray]] = None, *, where: Mask = True, dtype: DTypeLikeReals = None, constant: Optional[bool] = None*)

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates `log2(2**x1 + 2**x2)`. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

This docstring was adapted from that of numpy.logaddexp2 [1]

Parameters

x1, x2
[ArrayLike] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

logaddexp2
[Tensor] Base-2 logarithm of `2**x1 + 2**x2`.

See also:

`logaddexp`
Logarithm of the sum of exponentiations of the inputs.

References

[1]

Examples

```
>>> import mygrad as mg
>>> prob1 = mg.log2(1e-50)
>>> prob2 = mg.log2(2.5e-50)
>>> prob12 = mg.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(Tensor(-166.09640474436813), Tensor(-164.77447664948076), Tensor(-164.
↪28904982231052))
>>> 2 ** prob12
Tensor(3.4999999999999914e-50)
```

Attributes

signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.10.5 Other special functions

add_sequence(*variables[, constant])	`f(a, b, ...) -> a + b + ...`
multiply_sequence(*variables[, constant])	`f(a, b, ...) -> a * b * ...`
sinc(a, *[, constant])	`f(a) -> sin(a) / a`

mygrad.add_sequence

`mygrad.`**`add_sequence`**(**variables: ArrayLike*, *constant: Optional[bool] = None*) → Tensor

`f(a, b, ...) -> a + b + ...`

Add a sequence of tensors.

Parameters

variables
[ArrayLike] A sequence of broadcast-compatible tensors. Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

Notes

It is more efficient to back-propagate through this function than it is through a computational graph with N-1 corresponding addition operations.

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor([1. , 2.])
>>> y = mg.tensor([-1.])
>>> z = mg.tensor([[1.]])
>>> out = mg.add_sequence(x, y, z); out
    Tensor([[1., 2.]])
```

```
>>> out.backward()
>>> x.grad
array([1., 1.])
>>> y.grad
array([2.])
>>> z.grad
array([[2.]])
```

mygrad.multiply_sequence

mygrad.**multiply_sequence**(**variables: ArrayLike*, *constant: Optional[bool] = None*) → Tensor

`f(a, b, ...) -> a * b * ...`

Multiply a sequence of tensors.

Parameters

variables
[ArrayLike] A sequence of broadcast-compatible tensors. Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

Notes

It is more efficient to back-propagate through this function than it is through a computational graph with N-1 corresponding multiplication operations.

Examples

```
>>> import mygrad as mg
>>> x = mg.tensor([1. , 2.])
>>> y = mg.tensor([-1.])
>>> z = mg.tensor([[1.]])
>>> out = mg.multiply_sequence(x, y, z); out
    Tensor([[-1., -2.]])
```

```
>>> out.backward()
>>> x.grad
array([-1., -1.])
>>> y.grad
array([3.])
>>> z.grad
array([[-3.]])
```

mygrad.sinc

`mygrad.sinc`(*a: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

`f(a) -> sin(a) / a`

Parameters

a
[ArrayLike]

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor

3.10.6 Arithmetic operations

add(x1, x2[, out, where, dtype, constant])	Add the arguments element-wise.
reciprocal(x[, out, where, dtype, constant])	Return the reciprocal of the argument element-wise.
positive(x[, out, where, dtype, constant])	Returns a copy of the tensor.
negative(x[, out, where, dtype, constant])	Negates the tensor element-wise.
multiply(x1, x2[, out, where, dtype, constant])	Multiply the arguments element-wise.
divide	alias of `true_divide`
power(x1, x2[, out, where, dtype, constant])	First tensor elements raised to powers from second tensor, element-wise.
subtract(x1, x2[, out, where, dtype, constant])	Subtract the arguments element-wise.

mygrad.add

class `mygrad.add`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Add the arguments element-wise.

This docstring was adapted from that of numpy.add [1]

Parameters

x1, x2
[ArrayLike] The arrays to be added. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output). Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

add
[Tensor] The sum of *x1* and *x2*, element-wise.

Notes

Equivalent to *x1* + *x2* in terms of tensor broadcasting.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.add(1.0, 4.0)
Tensor(5.0)
>>> x1 = mg.tensor([[0., 1., 2.],
...                 [3., 4., 5.],
...                 [6., 7., 8.]])
>>> x2 = mg.tensor([0., 1., 2.])
>>> mg.add(x1, x2)
Tensor([[  0.,   2.,   4.],
        [  3.,   5.,   7.],
        [  6.,   8.,  10.]])
```

Attributes

signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.reciprocal

class `mygrad.`**`reciprocal`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Return the reciprocal of the argument element-wise.

This docstring was adapted from that of numpy.reciprocal [1]

Parameters

x

[ArrayLike] Input array.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

reciprocal
[Tensor]

Notes

Note: This function is not designed to work with integers.

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.reciprocal(2.)
Tensor(0.5)
>>> mg.reciprocal([1, 2., 3.33])
Tensor([ 1.        ,  0.5       ,  0.3003003])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.positive

class `mygrad.`**`positive`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Returns a copy of the tensor.

This docstring was adapted from that of numpy.positive [1]

Parameters

x

[ArrayLike] Input array.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

positive
[Tensor]

Notes

Equivalent to *x.copy()*, but only defined for types that support arithmetic.

References

[1]

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.negative

class mygrad.**negative**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Negates the tensor element-wise.

This docstring was adapted from that of numpy.negative [1]

Parameters

x
[ArrayLike or scalar] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

negative
[Tensor] The combination of *x1* and *x2*, element-wise.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.negative([1.,-1.])
Tensor([-1.,  1.])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.multiply

`class mygrad.multiply`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Multiply the arguments element-wise.

This docstring was adapted from that of numpy.multiply [1]

Parameters

x1, x2
[ArrayLike] Input arrays to be multiplied. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output). Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

multiply
[Tensor] The product of *x1* and *x2*, element-wise.

Notes

Equivalent to *x1* * *x2* in terms of tensor broadcasting.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.multiply(2.0, 4.0)
Tensor(8.0)
```

```
>>> x1 = mg.tensor([[0., 1., 2.],
...                 [3., 4., 5.],
...                 [6., 7., 8.]])
>>> x2 = mg.tensor([0., 1., 2.])
>>> mg.multiply(x1, x2)
Tensor([[  0.,   1.,   4.],
        [  0.,   4.,  10.],
        [  0.,   7.,  16.]])
```

Attributes

signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.divide

`mygrad.divide`

alias of `true_divide`

mygrad.power

class `mygrad.power`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

First tensor elements raised to powers from second tensor, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. Note that an integer type raised to a negative integer power will raise a ValueError.

This docstring was adapted from that of numpy.power [1]

Parameters

x1

[ArrayLike] The bases.

x2

[ArrayLike] The exponents. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output). Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

power
[Tensor] The combination of *x1* and *x2*, element-wise.

See also:

`float_power`
power function that promotes integers to float

References

[1]

Examples

Cube each element in a list.

```
>>> import mygrad as mg
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> mg.power(x1, 3)
Tensor([  0,   1,   8,  27,  64, 125])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> mg.power(x1, x2)
Tensor([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = mg.tensor([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
Tensor([[1, 2, 3, 3, 2, 1],
        [1, 2, 3, 3, 2, 1]])
>>> mg.power(x1, x2)
Tensor([[ 0,  1,  8, 27, 16,  5],
        [ 0,  1,  8, 27, 16,  5]])
```

Attributes

identity
signature

Methods

accumulate([axis, dtype, out, constant])	Not implemented
at(indices[, b, constant])	Not implemented
outer(b, *[, dtype, out])	Not Implemented
reduce([axis, dtype, out, keepdims, ...])	Not Implemented
reduceat(indices[, axis, dtype, out])	Not Implemented

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
accumulate([axis, dtype, out, constant])	Not implemented
at(indices[, b, constant])	Not implemented
outer(b, *[, dtype, out])	Not Implemented
reduce([axis, dtype, out, keepdims, ...])	Not Implemented
reduceat(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.subtract

class `mygrad.subtract`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Subtract the arguments element-wise.

This docstring was adapted from that of numpy.subtract [1]

Parameters

x1, x2
[ArrayLike] The arrays to be subtracted from each other. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output). Non-tensor array-likes are treated as constants.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

subtract
[Tensor] The difference of *x1* and *x2*, element-wise.

Notes

Equivalent to `x1 - x2` in terms of tensor broadcasting.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.subtract(1.0, 4.0)
Tensor(-3.0)
```

```
>>> x1 = mg.tensor([[0., 1., 2.],
...                 [3., 4., 5.],
...                 [6., 7., 8.]])
>>> x2 = mg.tensor([0., 1., 2.])
>>> mg.subtract(x1, x2)
Tensor([[ 0.,  0.,  0.],
        [ 3.,  3.,  3.],
        [ 6.,  6.,  6.]])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.10.7 Miscellaneous

`clip`(a, a_min, a_max[, out, constant])	Clip (limit) the values in an array.
`sqrt`(x[, out, where, dtype, constant])	The square root, elementwise.
`cbrt`(x[, out, where, dtype, constant])	The cube root elementwise.
`square`(x[, out, where, dtype, constant])	Return the square of the argument element-wise.
`absolute`(x[, out, where, dtype, constant, ...])	The absolute value, computed elementwise.
`maximum`(x1, x2[, out, where, dtype, constant])	Pair-wise maximum of tensor elements.
`minimum`(x1, x2[, out, where, dtype, constant])	Pair-wise minimum of tensor elements.

mygrad.clip

`mygrad.clip`(*a: ArrayLike*, *a_min: Optional[ArrayLike]*, *a_max: Optional[ArrayLike]*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *constant: Optional[bool] = None*) → Tensor

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to *mg.minimum(a_max, mg.maximum(a, a_min))*`.

No check is performed to ensure `a_min < a_max`.

This docstring was adapted from that of *numpy.clip*

Parameters

a
[ArrayLike] Array containing elements to clip.

a_min
[Optional[float, ArrayLike]] Minimum value. If *None*, clipping is not performed on lower interval edge. Not more than one of *a_min* and *a_max* may be *None*.

a_max
[Optional[float, ArrayLike]] Maximum value. If *None*, clipping is not performed on upper interval edge. Not more than one of *a_min* and *a_max* may be *None*. If *a_min* or *a_max* are ArrayLike, then the three arrays will be broadcasted to match their shapes.

out
[Optional[Union[ndarray, Tensor]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

Tensor
A tensor with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

Examples

```
>>> import mygrad as mg
>>> a = mg.arange(10)
>>> mg.clip(a, 1, 8)
Tensor([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
Tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> mg.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
Tensor([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

mygrad.sqrt

class `mygrad.sqrt`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

The square root, elementwise.

This docstring was adapted from that of numpy.sqrt [1]

Parameters

x
[ArrayLike] The values whose square-roots are required.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it

must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[ndarray] A tensor of the same shape as x, containing the positive square-root of each element in x. Negative-valued inputs produce nans.

Notes

sqrt has–consistent with common convention–as its branch cut the real “interval” [*-inf*, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.sqrt([1, 4, 9])
Tensor([ 1.,  2.,  3.])
```

```
>>> mg.sqrt([4, -1, mg.inf])
Tensor([ 2., nan, inf])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.cbrt

class `mygrad.`**`cbrt`**(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

The cube root elementwise.

This docstring was adapted from that of numpy.cbrt [1]

Parameters

x

[ArrayLike] The values whose cube-roots are computed.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[ndarray] A tensor of the same shape as *x*, containing the cube cube-root of each element in *x*.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.cbrt([1, 8, 27])
Tensor([ 1.,  2.,  3.])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.square

class `mygrad.square`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, *, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Return the square of the argument element-wise.

This docstring was adapted from that of numpy.square [1]

Parameters

x
[ArrayLike] Input data.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it

must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

where

[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

Returns

square

[Tensor]

See also:

sqrt
power

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.square([100., 1000.])
array([10.,  100.])
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.absolute

class `mygrad.absolute`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*, *nan_to_num: bool = True*)

The absolute value, computed elementwise.

This docstring was adapted from that of numpy.absolute [1]

Parameters

x
[ArrayLike] Input array.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

nan_to_num
[bool, optional (default=True)] If *True* then gradients that would store nans due to the presence of zeros in *x* will instead store zeros in those places.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

absolute
[Tensor] An ndarray containing the absolute value of each element in *x*.

References

[1]

Examples

```
>>> import mygrad as mg
>>> x = mg.array([-1.2, 1.2])
>>> mg.absolute([-1.2, 1.2])
Tensor([ 1.2,  1.2])
```

The absolute-value function is not differentiable at *x=0.0*. By default the derivative at this point is treated as 0.

```
>>> x = mg.tensor([-2.0, 0.0, 2.0])
>>> mg.absolute(x).backward()
>>> x.grad
np.array([-1., 0., 1.])
```

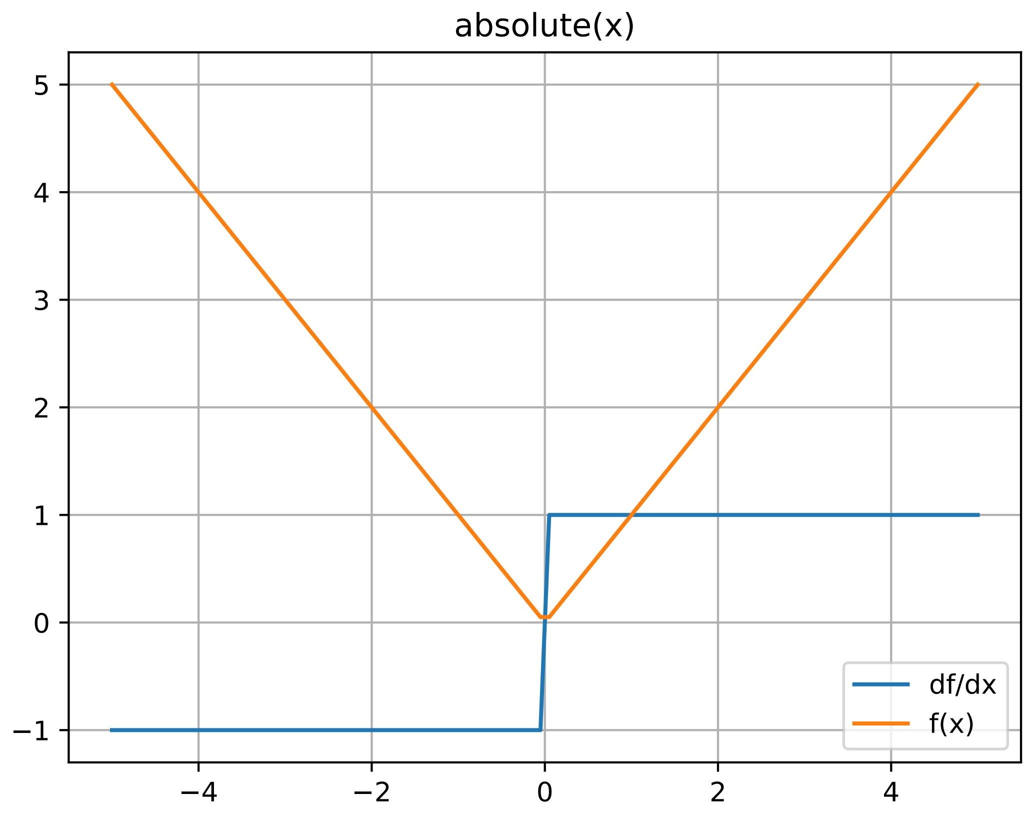
However a more rigorous behavior can be enabled such that the undefined derivative will be returned as *nan*.

```
>>> x = mg.tensor([-2.0, 0.0, 2.0])
>>> mg.absolute(x, nan_to_num=False).backward()
>>> x.grad
np.array([-1., nan, 1.])
```

Plot the function and its derivate over `[-10, 10]`:

Attributes

identity
signature


absolute(x)
df/dx
f(x)
5
4
3
2
1
0
−1
−4
−2
0
2
4

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.maximum

class `mygrad.`**`maximum`**(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Pair-wise maximum of tensor elements.

This docstring was adapted from that of numpy.maximum [1]

Parameters

x1, x2

[ArrayLike] The tensors holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The maximum of *x1* and *x2*, element-wise.

See also:

`minimum`
Element-wise minimum of two arrays, propagates NaNs.

Notes

The maximum is equivalent to `mg.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are nans, but it is faster and does proper broadcasting.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.maximum([2, 3, 4], [1, 5, 2])
Tensor([2, 5, 4])
```

```
>>> mg.maximum(mg.eye(2), [0.5, 2]) # broadcasting
Tensor([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> mg.maximum([mg.nan, 0, mg.nan], [0, mg.nan, mg.nan])
Tensor([nan, nan, nan])
>>> mg.maximum(mg.Inf, 1)
Tensor(inf)
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

mygrad.minimum

class `mygrad.minimum`(*x1: ArrayLike*, *x2: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Pair-wise minimum of tensor elements.

This docstring was adapted from that of numpy.minimum [1]

Parameters

x1, x2
[ArrayLike] The tensors holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The minimum of *x1* and *x2*, element-wise.

See also:

`maximum`
Element-wise maximum of two arrays, propagates NaNs.

Notes

The minimum is equivalent to `mg.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

References

[1]

Examples

```
>>> import mygrad as mg
>>> mg.minimum([2, 3, 4], [1, 5, 2])
Tensor([1, 3, 2])
```

```
>>> mg.minimum(mg.eye(2), [0.5, 2]) # broadcasting
Tensor([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> mg.minimum([mg.nan, 0, mg.nan],[0, mg.nan, mg.nan])
Tensor([nan, nan, nan])
>>> mg.minimum(-mg.Inf, 1)
Tensor(-inf)
```

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.11 Indexing Routines (`mygrad.indexing_routines`)

3.11.1 Generating index tensors

where(condition, [x, y])	Return elements chosen from *x* or *y* depending on *condition*.

mygrad.where

`mygrad.`**`where`**(*condition*[, *x*, *y*])

Return elements chosen from *x* or *y* depending on *condition*.

Note: When only `condition` is provided, this function is a shorthand for `np.asarray(condition).nonzero()`. The rest of this documentation covers only the case where all three arguments are provided.

This docstring was adapted from that of `numpy.where`.

Parameters

condition
[ArrayLike, bool] Where True, yield *x*, otherwise yield `y`. `x`, `y` and *condition* need to be broadcastable to some shape.

x
[ArrayLike] Values from which to chosen where `condition` is `True`.

y
[ArrayLike] Values from which to chosen where `condition` is `False`.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

out

[mygrad.Tensor] A tensor with elements from *x* where *condition* is True, and elements from *y* elsewhere.

Examples

```
>>> import mygrad as mg
>>> a = mg.arange(10)
>>> a
Tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> mg.where(a < 5, a, 10*a)
Tensor([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

This can be used on multidimensional tensors too:

```
>>> mg.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
Tensor([[1, 8],
        [3, 4]])
```

The shapes of x, y, and the condition are broadcast together:

```
>>> x, y = np.ogrid[:3, :4]
>>> mg.where(x < y, x, 10 + y)  # both x and 10+y are broadcast
Tensor([[10,  0,  0,  0],
        [10, 11,  1,  1],
        [10, 11, 12,  2]])
```

```
>>> a = mg.Tensor([[0, 1, 2],
...                [0, 2, 4],
...                [0, 3, 6]])
>>> mg.where(a < 4, a, -1)  # -1 is broadcast
Tensor([[ 0,  1,  2],
        [ 0,  2, -1],
        [ 0,  3, -1]])
```

3.12 Neural network operations (`mygrad.nnet`)

3.12.1 Layer operations

batchnorm(x, *[, gamma, beta, constant])	Performs batch normalization on `x`.
conv_nd(x, filter_bank, *, stride[, ...])	Use `filter_bank` (`w`) to perform strided N-dimensional neural network-style convolutions (see Notes) over `x`..
max_pool(x, pool, stride, *[, constant])	Perform max-pooling over the last N dimensions of a data batch.
gru(X, Uz, Wz, bz, Ur, Wr, br, Uh, Wh, bh[, ...])	Performs a forward pass of sequential data through a Gated Recurrent Unit layer, returning the 'hidden-descriptors' arrived at by utilizing the trainable parameters as follows.

mygrad.nnet.layers.batchnorm

`mygrad.nnet.layers.`**`batchnorm`**(*x: ArrayLike*, ***, *gamma: Optional[ArrayLike] = None*, *beta: Optional[ArrayLike] = None*, *eps: float*, *constant: Optional[bool] = None*) → Tensor

Performs batch normalization on `x`:

```
y(x) = (x - E[x]) / sqrt(Var[x] + eps)
batchnorm(x) = gamma * y(x) + beta
```

Where $E[x]$ and $Var[x]$ represent the mean and variance, respectively, over axis-1 of `x`. The subsequent affine transformation on `y` is optional.

Parameters

x
[array_like, shape=(N, C, …)] The batch to be normalized within each entry of C

gamma
[Optional[array_like], shape=(C,)] Optional per-channel scaling factors to be applied after the normalization step.

beta
[Optional[array_like], shape=(C,)] Optional per-channel scaling bias factors to be applied after the normalization step.

eps
[Real] A small non-negative number.

constant
[bool, optional (default=False)] If True, the resulting Tensor is a constant.

Returns

mygrad.Tensor
The batch-normalized data.

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import batchnorm
>>> x = mg.Tensor([1., 4., 1.]).reshape(3, 1)
>>> batchnorm(x, eps=0)
Tensor([[-0.70710678],
        [ 1.41421356],
        [-0.70710678]])
```

mygrad.nnet.layers.conv_nd

`mygrad.nnet.layers.conv_nd`(*x: ArrayLike*, *filter_bank: ArrayLike*, ***, *stride: Union[int, Tuple[int, ...]]*, *padding: Union[int, Tuple[int, ...]] = 0*, *dilation: Union[int, Tuple[int, ...]] = 1*, *constant: Optional[bool] = None*) → Tensor

Use `filter_bank` (`w`) to perform strided N-dimensional neural network-style convolutions (see Notes) over `x`.:

```
f(x, w) -> x  w

shapes:
(N, C, X0, ...)  (F, C, W0, ...) -> (N, F, G0, ...)
```

`x` represents a batch of data over which the filters are convolved. Specifically, it must be a tensor of shape $(N, C, X_0, ...)$, where N is the number of samples in the batch, C is the channel-depth of each datum, and $(X_0, ...)$ are the dimensions over which the filters are convolved. Accordingly, each filter must have a channel depth of C.

Thus convolving F filters, each with a shape $(C, W_0, ...)$, over the data batch will produce a tensor of shape $(N, F, G_0, ...)$, where $(G_0, ...)$ is the shape of the grid commensurate with the filter placements

Parameters

x
[ArrayLike, shape=(N, C, Xo, . . .)] The data batch to be convolved over.

filter_bank
[Union[Tensor, array_like], shape=(F, C, Wo, . . .)] The filters used to perform the convolutions.

stride
[Union[int, Tuple[int, . . .]]] (keyword-only argument) The step-size with which each filter is placed along the H and W axes during the convolution. The tuple indicates (stride-0, . . .). If a single integer is provided, this stride is used for all convolved dimensions

padding
[Union[int, Tuple[int, . . .]]] (keyword-only argument) The number of zeros to be padded to both ends of each convolved dimension, respectively. If a single integer is provided, this padding is used for all of the convolved axes

dilation
[Union[int, Tuple[int, . . .]], optional (default=1)] (keyword-only argument) The spacing used when placing kernel elements along the data. E.g. for a 1D convolution the ith placement of the kernel multiplied against the dilated-window: `x[:, :, i*s:(i*s + w*d):d]`, where `s` is the stride, `w` is the kernel-size, and `d` is the dilation factor.

If a single integer is provided, that dilation value is used for all of the convolved axes

constant
[Optional[None]] If True, the resulting Tensor is a constant.

Returns

Tensor, shape=(N, F, G0, ...)
The result of each filter being convolved over each datum in the batch.

Notes

- The filters are *not* flipped by this operation, meaning that an auto-correlation is being performed rather than a true convolution.
- Only 'valid' filter placements – where the filters overlap completely with the (padded) data – are permitted.

Examples

Here we perform a 1D convolution of a constant-valued kernel, k, with a 'square-wave' signal, x, using stride-1. Note that because we are constrained to doing deep learning-style convolutions, that we prepend the dimensions $(N = 1, C = 1)$ to x, and $(F = 1, C = 1)$ and to k. That is, we are performing a convolution on one, single-channeled signal using one kernel.

See that this convolution produces the expected triangle-shaped response. The shape of the resulting tensor is $(N = 1, F = 1, G_0 = 12)$. That is, the length-5 kernel can be placed in 12 valid positions, using a stride of 1.

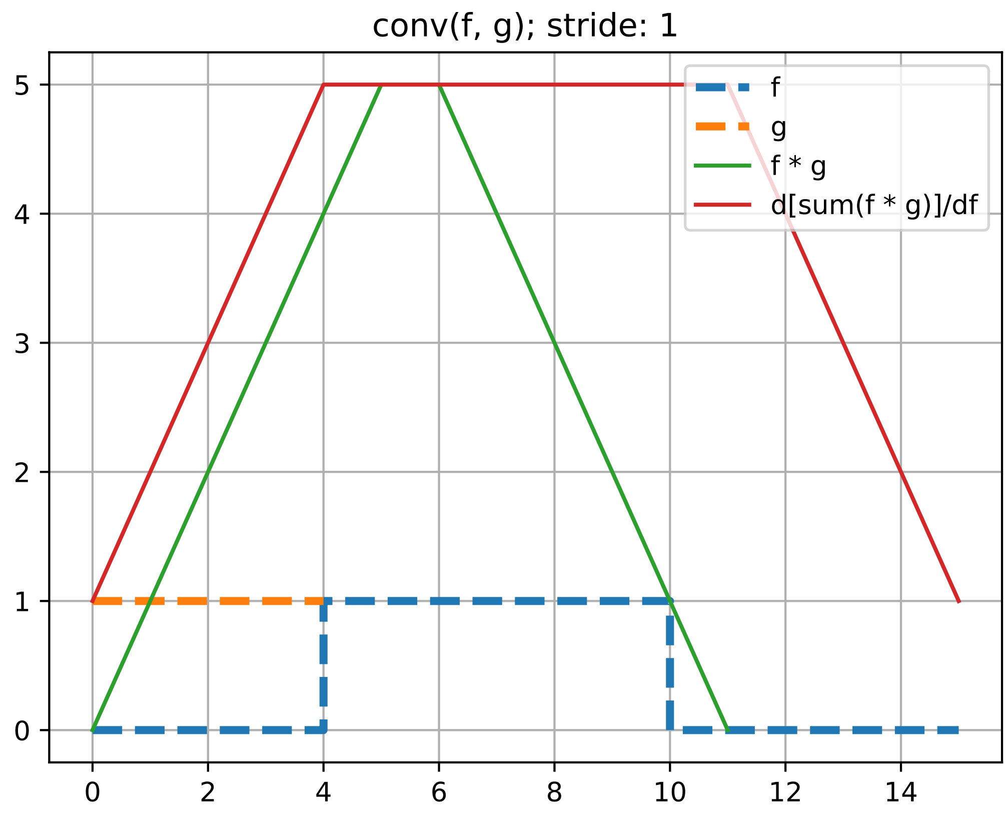
```
>>> import mygrad as mg
>>> from mygrad.nnet import conv_nd
>>> x = mg.zeros((1, 1, 16))  # a square-wave signal
>>> x[..., 5:11] = 1
>>> k = mg.ones((1, 1, 5))    # a constant-valued kernel
>>> conv_nd(x, k, stride=1)   # performing a stride-1, 1D convolution
Tensor([[[0., 1., 2., 3., 4., 5., 5., 4., 3., 2., 1., 0.]]], dtype=float32)
```

Back-propagating through the (summed) convolution:

```
>>> conv_nd(x, k, stride=1).sum().backward()  # sum to a scalar to perform back-prop
>>> x.grad  # d(summed_conv)/dx
array([[[1., 2., 3., 4., 5., 5., 5., 5., 5., 5., 5., 5., 4., 3., 2., 1.]]],
      dtype=float32)
>>> k.grad  # d(summed_conv)/dk
array([[[6., 6., 6., 6., 6.]]])
```

Let's apply a edge-detection kernel to each color channel of an RGB image.

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.image as mpimg
>>> from mygrad.nnet.layers import conv_nd
>>> # A shape-(H, W, 3) RGB image
>>> img = mpimg.imread('../_static/meerkat.png')
>>> # We'll treat this like a batch of three greyscale images
>>> # where each "image" is actually a color channel
>>> # shape-(H, W, 3) -> shape-(3, 1, H, W)
>>> x = img.transpose(2, 0, 1)[:, None, :, :]
```

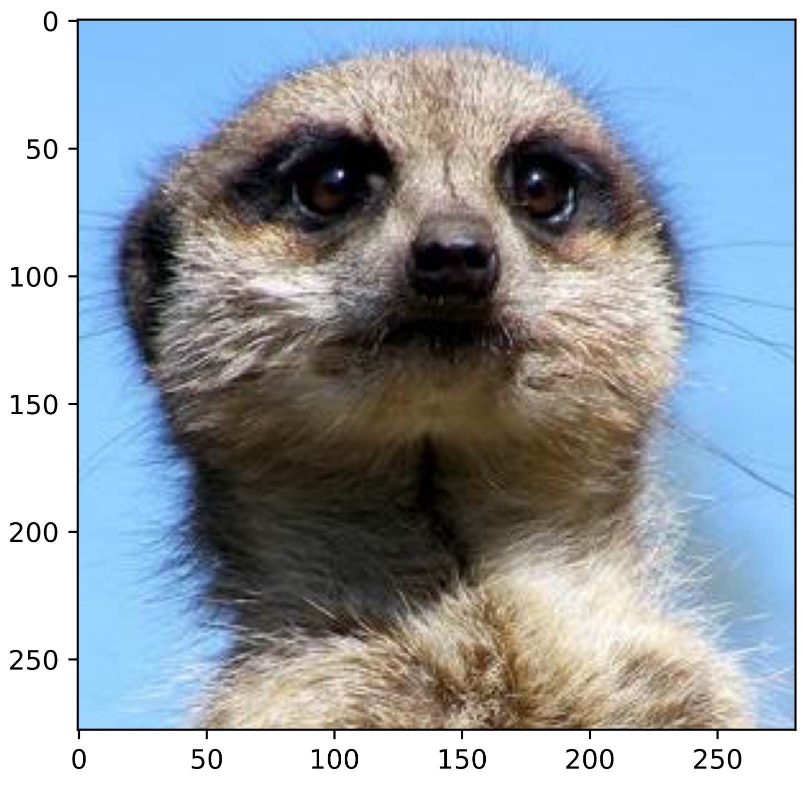

conv(f, g); stride: 1
f
g
f * g
d[sum(f * g)]/df
0
1
2
3
4
5
0
2
4
6
8
10
12
14

```
>>> # edge detection kernel
>>> kernel = np.array([[-1, -1, -1],
...                    [-1,  8, -1],
...                    [-1, -1, -1]])
>>> # (Hf, Wf) --> (1, 1, Hf, Wf)
>>> kernel = kernel.reshape(1, 1, *kernel.shape)
```

```
>>> # conv: (3, 1, H, W) w/ (1, 1, Hf, Wf) --> (3, 1, H', W')
>>> # squeeze + transpose: (3, 1, H', W') --> (H', W', 3)
>>> processed = conv_nd(x, kernel, stride=(1, 1))
>>> processed = processed.data.squeeze().transpose(1, 2, 0)
```

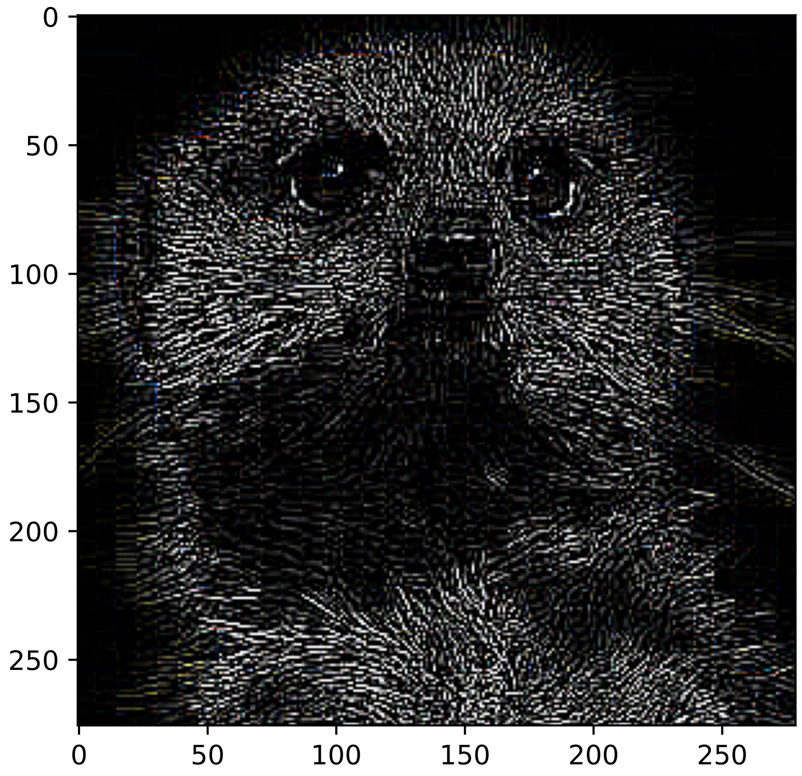
```
>>> fig, ax = plt.subplots()
>>> ax.imshow(img)
```

```
>>> fig, ax = plt.subplots()
>>> ax.imshow(processed)
```



Now, let's demonstrate a more typical usage for `conv_nd` in the context of neural networks. `x` will represent 10, 32x32 RGB images, and we will use 5 distinct 2x2 kernels to convolve over each of these images . Note that each kernel must possess 3-channel - one for each RGB channel.

That is, we will be performing NxF channel-wise 2D convolutions. Supposing that we don't want the kernel


0
50
100
150
200
250
0
50
100
150
200
250

placements to overlap, we can use a stride of 2. In total, this will produce a shape-$(N = 10, F = 5, G_0 = 16, G_1 = 16)$ tensor as a result.

```
>>> import mygrad as mg
>>> x = mg.random.rand(10, 3, 32, 32))  # creating 10 random 32x32 RGB images
>>> k = mg.random.rand(5, 3, 2, 2))     # creating 5 random 3-channel 2x2 kernels
```

Given the shapes of `x` and `k`, `conv_nd` automatically executes a 2D convolution:

```
>>> conv_nd(x, k, stride=2).shape
(10, 5, 16, 16)
```

Extrapolating further, `conv_nd` is capable of performing ND convolutions!

Performing a convolution over a batch of single-channel, "spatial-3D" tensor data:

```
>>> # shape-(N=1, C=1, X=10, Y=12, Z=10)
>>> x = mg.random.rand(1, 1, 10, 12, 10)
>>> # shape-(F=2, C=1, Wx=3, Wy=1, Wz=2)
>>> k = mg.random.rand(2, 1, 3, 1, 32)
>>> conv_nd(x, k, stride=1).shape
(1, 2, 8, 12, 9)
```

mygrad.nnet.layers.max_pool

`mygrad.nnet.layers.max_pool`(*x: ArrayLike*, *pool: Tuple[int, ...]*, *stride: Union[int, Tuple[int, ...]]*, ***, *constant: Optional[bool] = None*) → Tensor

Perform max-pooling over the last N dimensions of a data batch.

The data consists of N trailing axes to be pooled over, denoted by `C0, ...`. These can be preceded, optionally, by un-pooled axes, denoted by `(N0, ...)`. The dimensions of the window over which pooling is performed is denoted by `P0, ...`. The window is placed with stride values `S0, ...`.

Ultimately the pooled channels have a shape `G0, ...`.

Parameters

x
[mygrad.Tensor, shape=([...], C0, ...)] The data batch; to be pooled along the trailing axes denoted by `C0, ...`.

pool
[Tuple[Integral, ...], (P0, ...)] The extent of the pooling window along the `(C0, ...)` axes, respectively. The length of *pool* determines `N` - the number of trailing dimensions to pool over.

stride
[Union[Integral, Tuple[Integral, ...]], (S0, ...)] The spacing used to place the pooling window, along `(P0, ...)` axes, respectively. If a single value is provided, it is used for all `N` pooling axes.

constant
[Optional[None]] If True, the resulting Tensor is a constant.

Returns

——-

Tensor, shape=([...], G0, ...)
The pooled data batch.

Notes

Only "valid" placements of the pooling window are permitted - the pooling window cannot extend passed the "boundaries" of the data dimensions.

Examples

Simple 2D pooling on a 2D tensor. Tiling a 2x2 max-pool window with stride-1 over a shape-(3, 3) tensor x:

```
>>> import  mygrad as mg
>>> from mygrad.nnet import max_pool
>>> x = mg.Tensor([[0., 10.,  8.],
...                [2.,  7.,  3.],
...                [5.,  7., 20.]])
>>> out = max_pool(x, pool=(2, 2), stride=1)
>>> out
Tensor([[ 10., 10.],
        [  7., 20.]])
>>> out.sum().backward()  # sum to reduce to scalar for back-prop
>>> x.grad  # dout/dx
array([[0., 2., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Let's perform 1D pooling on a 2D tensor. Each row of the tensor will be pooled over independently. Let's apply a size-2 max-pool window to each row of x, using a stride of 1:

```
>>> x = mg.Tensor([[0., 10., 8.],
...                [9., 7.,  3.],
...                [5., 0., 20.]])
>>> max_pool(x, pool=(2,), stride=1)
Tensor([[10., 10.],
        [ 9.,  7.],
        [ 5., 20.]])
```

Here we perform pooling over the trailing two dimensions of a 4D tensor, x. By specifying pool = (2, 2), we instruct max_pool to tile a 2x2 pooling window along these last two axes. Let's apply the window every two rows, and for each column; i.e. we specify stride = (2, 1):

```
>>> import numpy as np
>>> x = mg.Tensor(np.random.rand(10, 3, 12, 12))
>>> pool = (2, 2)   # 2x2 pooling over the last axes
>>> stride = (2, 1) # Apply 2x1 stride
>>> out = max_pool(x, pool, stride)  # max-pooled Tensor
>>> out.shape
(10, 3, 6, 11)
```

Had we specified, say, pool = (3, 2, 2), then a 3x2x2 pooling window would have been tiled along the last *three* axes of x.

mygrad.nnet.layers.gru

`mygrad.nnet.layers.gru`(*X*, *Uz*, *Wz*, *bz*, *Ur*, *Wr*, *br*, *Uh*, *Wh*, *bh*, *s0=None*, *bp_lim=None*, *dropout=0.0*, *constant=None*)

Performs a forward pass of sequential data through a Gated Recurrent Unit layer, returning the 'hidden-descriptors' arrived at by utilizing the trainable parameters as follows:

```
Z_{t} = sigmoid(X_{t} Uz + S_{t-1} Wz + bz)
R_{t} = sigmoid(X_{t} Ur + S_{t-1} Wr + br)
H_{t} =    tanh(X_{t} Uh + (R{t} * S_{t-1}) Wh + bh)
S_{t} = (1 - Z{t}) * H{t} + Z{t} * S_{t-1}
```

Parameters

X
[array_like, shape=(T, N, C)] The sequential data to be passed forward.

Uz
[array_like, shape=(C, D)] The weights used to map sequential data to its hidden-descriptor representation

Wz
[array_like, shape=(D, D)] The weights used to map a hidden-descriptor to a hidden-descriptor.

bz
[array_like, shape=(D,)] The biases used to scale a hidden-descriptor.

Ur
[array_like, shape=(C, D)] The weights used to map sequential data to its hidden-descriptor representation

Wr
[array_like, shape=(D, D)] The weights used to map a hidden-descriptor to a hidden-descriptor.

br
[array_like, shape=(D,)] The biases used to scale a hidden-descriptor.

Uh
[array_like, shape=(C, D)] The weights used to map sequential data to its hidden-descriptor representation

Wh
[array_like, shape=(D, D)] The weights used to map a hidden-descriptor to a hidden-descriptor.

bh
[array_like, shape=(D,)] The biases used to scale a hidden-descriptor.

s0
[Optional[array_like], shape=(N, D)] The 'seed' hidden descriptors to feed into the RNN. If None, a Tensor of zeros of shape (N, D) is created.

bp_lim
[Optional[int]] *This feature is experimental and is currently untested*. The (non-zero) limit of the depth of back propagation through time to be performed. If *None* back propagation is passed back through the entire sequence.

E.g. *bp_lim=3* will propagate gradients only up to 3 steps backward through the recursive sequence.

dropout
[float (default=0.), 0 <= dropout < 1] If non-zero, the dropout scheme described in [1] is applied. See Notes for more details.

constant
[bool, optional (default=False)] If True, the resulting Tensor is a constant.

Returns

mygrad.Tensor, shape=(T+1, N, D)
The sequence of 'hidden-descriptors' produced by the forward pass of the RNN.

Notes

- T : Sequence length
- N : Batch size
- C : Length of single datum
- D : Length of 'hidden' descriptor

The GRU system of equations is given by:

$$\begin{aligned}Z_t &= \sigma(X_t U_z + S_{t-1} Wz + bz)\\ R_t &= \sigma(X_t U_r + S_{t-1} Wr + br)\\ H_t &= tanh(X_t U_h + (R_t * S_{t-1}) W_h + b_h)\\ S_t &= (1 - Z_t) * H_t + Z_t * S_{t-1}\end{aligned}$$

Following the dropout scheme specified in [1], the hidden-hidden weights (Wz/Wr/Wh) randomly have their weights dropped prior to forward/back-prop. The input connections (via Uz/Ur/Uh) have variational dropout ([2]) applied to them with a common dropout mask across all t. That is three static dropout masks, each with shape-(N,D), are applied to

$$\begin{aligned}X_t U_z\\ X_t U_r\\ X_t U_h\end{aligned}$$

respectively, for all t.

References

[1], [2]

3.12.2 Losses

`focal_loss`(class_probs, targets, *[, alpha, ...])	Return the per-datum focal loss.
`margin_ranking_loss`(x1, x2, y, margin, *[, ...])	Computes the margin average margin ranking loss. Equivalent to::.
`multiclass_hinge`(x, y_true[, hinge, constant])	Computes the average multiclass hinge loss.
`negative_log_likelihood`(x, y_true, *[, ...])	Returns the (weighted) negative log-likelihood loss between log-probabilities and y_true.
`softmax_crossentropy`(x, y_true, *[, constant])	Given the classification scores of C classes for N pieces of data,
`softmax_focal_loss`(scores, targets, *[, ...])	Applies the softmax normalization to the input scores before computing the per-datum focal loss.

mygrad.nnet.losses.focal_loss

`mygrad.nnet.losses.`**`focal_loss`**(*class_probs: ArrayLike*, *targets: ArrayLike*, ***, *alpha: float = 1*, *gamma: float = 0*, *constant: Optional[bool] = None*) → Tensor

Return the per-datum focal loss.

Parameters

class_probs
[ArrayLike, shape=(N, C)] The C class probabilities for each of the N pieces of data. Each value is expected to lie on (0, 1]

targets
[ArrayLike, shape=(N,)] The correct class indices, in [0, C), for each datum.

alpha
[Real, optional (default=1)] The weighting factor in the loss formulation.

gamma
[Real, optional (default=0)] The focusing parameter. Note that for =0 and =1, this is cross-entropy loss. Must be a non-negative value.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor, shape=(N,)
The per-datum focal loss.

Notes

The formulation for the focal loss introduced in https://arxiv.org/abs/1708.02002. It is given by -(1-p)log(p).

The focal loss for datum-i is given by

$$-\alpha\hat{y}_i(1-p_i)^\gamma\log(p_i)$$

where $\hat{y}_i$ is one in correspondence to the label associated with the datum and 0 elsewhere. That is, if the label y_k is 2 and there are four possible label values, then $\hat{y}_k = (0, 0, 1, 0)$.

It is recommended in the paper that you normalize by the number of foreground samples.

mygrad.nnet.losses.margin_ranking_loss

`mygrad.nnet.losses.`**`margin_ranking_loss`**(*x1: ArrayLike*, *x2: ArrayLike*, *y: ArrayLike*, *margin: float*, ***, *constant: Optional[bool] = None*) → Tensor

Computes the margin average margin ranking loss. Equivalent to:

```
>>> import mygrad as mg
>>> mg.mean(mg.maximum(0, margin - y * (x1 - x2)))
```

Parameters

x1
[ArrayLike, shape=(N,) or (N, D)] A batch of scores or descriptors to compare against those in *x2*

x2
[ArrayLike, shape=(N,) or (N, D)] A batch of scores or descriptors to compare against those in *x1*

y
[Union[int, ArrayLike], scalar or shape=(N,)] 1 or -1. Specifies whether the margin is compared against *(x1 - x2)* or *(x2 - x1)*, for each of the N comparisons.

margin
[float] A non-negative value to be used as the margin for the loss.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not backpropagate a gradient)

Returns

mygrad.Tensor, shape=()
The mean margin ranking loss.

mygrad.nnet.losses.multiclass_hinge

`mygrad.nnet.losses.`**`multiclass_hinge`**(*x: ArrayLike*, *y_true: ArrayLike*, *hinge: float = 1.0*, ***, *constant: Optional[bool] = None*) → Tensor

Computes the average multiclass hinge loss.

Parameters

x
[ArrayLike, shape=(N, K)] The K class scores for each of the N pieces of data.

y_true
[ArrayLike, shape=(N,)] The correct class-indices, in [0, K), for each datum.

hinge
[float] The size of the "hinge" outside of which a nonzero loss is incurred.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not backpropagate a gradient)

Returns

Tensor, shape-() (scalar)
The average multiclass hinge loss

Raises

TypeError
y_true must be an integer-type array-like object

ValueError
x must be a 2-dimensional array-like object *y_true* must be a shape-(N,) array-like object

mygrad.nnet.losses.negative_log_likelihood

`mygrad.nnet.losses.negative_log_likelihood`(*x: ArrayLike*, *y_true: ArrayLike*, ***, *weights: Optional[ArrayLike] = None*, *constant: Optional[bool] = None*) → Tensor

Returns the (weighted) negative log-likelihood loss between log-probabilities and y_true.

Note that this does not compute a softmax, so you should input log-probabilities to this. See `softmax_crossentropy` if you need your loss to compute a softmax.

Parameters

x
[ArrayLike, shape=(N, C)] The C log-probabilities for each of the N pieces of data.

y_true
[ArrayLike, shape=(N,)] The correct class indices, in [0, C), for each datum.

weights
[ArrayLike, shape=(C,) optional (default=None)] The weighting factor to use on each class, or None.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor, shape=()
The average (weighted) negative log-likelihood loss.

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import negative_log_likelihood
```

Let's take a simple case where N=1, and C=3. We'll thus make up classification scores for a single datum. Suppose the scores are identical for the three classes and that the true class is class-0, so that the log-probs are each 1/3:

```
>>> logprob = mg.log(1 / 3).item()
>>> x = mg.Tensor([[logprob, logprob, logprob]])  # a shape-(1, 3) tensor of log-
↪probabilities
>>> y_true = mg.Tensor([0])  # the correct class for this datum is class-0
>>> negative_log_likelihood(x, y_true)
Tensor(1.09861229)
```

Log-probabilities where the prediction is highly-confident and correct:

```
>>> x = mg.Tensor([[0, -20, -20]])
>>> negative_log_likelihood(x, y_true)
Tensor(0.)
```

Adding a class-weighting:

```
>>> x = mg.Tensor([[-4.6, -4.6, -0.02]])
>>> weights = mg.Tensor([2, 1, 1])
>>> negative_log_likelihood(x, y_true, weights=weights)
Tensor(9.2)
```

mygrad.nnet.losses.softmax_crossentropy

`mygrad.nnet.losses.softmax_crossentropy`(*x: ArrayLike*, *y_true: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

Given the classification scores of C classes for N pieces of data,

computes the NxC softmax classification probabilities. The cross entropy is then computed by using the true classification labels.

log-softmax is used for improved numerical stability.

Parameters

x
[ArrayLike, shape=(N, C)] The C class scores for each of the N pieces of data.

y_true
[ArrayLike, shape=(N,)] The correct class-indices, in [0, C), for each datum.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

The average softmax loss

Raises

ValueError
Bad dimensionalities for `x` or `y_true`

Notes

- N is the number of samples in the batch.
- C is the number of possible classes for which scores are provided.

Given the shape-(N, C) tensor of scores, `x`, the softmax classification probabilities are computed. That is, the score for class-k of a given datum (s_k) is normalized using the 'softmax' transformation:

$$p_k = \frac{e^{s_k}}{\sum_{i=1}^{C} e^{s_i}}$$

This produces the "prediction probability distribution", p, for each datum. The cross-entropy loss for that datum is then computed according to the true class-index for that datum, as reported in `y_true`. That is the "true probability distribution", t, for the datum is 1 for the correct class-index and 0 elsewhere.

The cross-entropy loss for that datum is thus:

$$l = -\sum_{k=1}^{C} t_k \log p_k$$

Having computed each per-datum cross entropy loss, this function then returns the loss averaged over all N pieces of data:

$$L = \frac{1}{N}\sum_{i=1}^{N} l_i$$

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import softmax_crossentropy
```

Let's take a simple case where N=1, and C=3. We'll thus make up classification scores for a single datum. Suppose the scores are identical for the three classes and that the true class is class-0:

```
>>> x = mg.Tensor([[2., 2., 2.]])  # a shape-(1, 3) tensor of scores
>>> y_true = mg.Tensor([0])  # the correct class for this datum is class-0
```

Because the scores are identical for all three classes, the softmax normalization will simply produce $p = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. Because class-0 is the "true" class, $t = [1., 0., 0.]$. Thus our softmax cross-entropy loss should be:

$$-(1 \times \log\frac{1}{3} + 0 \times \log\frac{1}{3} + 0 \times \log\frac{1}{3}) = \log(3) \approx 1.099$$

Let's see that this is what `softmax_crossentropy` returns:

```
>>> softmax_crossentropy(x, y_true)
Tensor(1.09861229)
```

Similarly, suppose a datum's scores are $[0, 0, 10^6]$, then the softmax normalization will return $p \approx [0., 0., 1.]$. If the true class for this datum is class-2, then the loss should be nearly 0, since p and t are essentially identical:

$$-(0 \times \log 0 + 0 \times \log 0 + 1 \times \log 1) = -\log(1) = 0$$

Now, let's construct `x` and `y_true` so that they incorporate the scores/labels for both of the data that we have considered:

```
>>> x = mg.Tensor([[2., 2.,  2.],  # a shape-(2, 3) tensor of scores
...                [0., 0., 1E6]])
>>> y_true = mg.Tensor([0, 2])     # the class IDs for the two data
```

`softmax_crossentropy(x, y_true)` will return the average loss of these two data, $\frac{1}{2}(1.099 + 0) \approx 0.55$:

```
>>> softmax_crossentropy(x, y_true)
Tensor(0.54930614)
```

mygrad.nnet.losses.softmax_focal_loss

mygrad.nnet.losses.**softmax_focal_loss**(*scores: ArrayLike*, *targets: ArrayLike*, ***, *alpha: float = 1*, *gamma: float = 0*, *constant: Optional[bool] = None*) → Tensor

Applies the softmax normalization to the input scores before computing the per-datum focal loss.

Parameters

scores
[ArrayLike, shape=(N, C)] The C class scores for each of the N pieces of data.

targets
[ArrayLike, shape=(N,)] The correct class indices, in [0, C), for each datum.

alpha
[Real, optional (default=1)] The weighting factor in the loss formulation.

gamma
[Real, optional (default=0)] The focusing parameter. Note that for =0 and =1, this is cross-entropy loss. Must be a non-negative value.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor, shape=(N,)
The per-datum focal loss.

Notes

The formulation for the focal loss introduced in https://arxiv.org/abs/1708.02002. It is given by -(1-p)log(p).

The focal loss for datum-i is given by

$$-\alpha\hat{y}_i(1-p_i)^\gamma\log(p_i)$$

where $\hat{y}_i$ is one in correspondence to the label associated with the datum and 0 elsewhere. That is, if the label y_k is 2 and there are four possible label values, then $\hat{y}_k = (0, 0, 1, 0)$.

It is recommended in the paper that you normalize by the number of foreground samples.

3.12.3 Activations

elu(x, alpha, *[, constant])	Returns the exponential linear activation (ELU) elementwise along x.
glu(x[, axis, constant])	Returns the Gated Linear Unit A * (B), where A and B are split from *x*.
hard_tanh(x, *[, lower_bound, upper_bound, ...])	Returns the hard hyperbolic tangent function.
leaky_relu(x, slope, *[, constant])	Returns the leaky rectified linear activation elementwise along x.
logsoftmax(x[, axis, constant])	Applies the log-softmax activation function.
selu(x, *[, constant])	Returns the scaled exponential linear activation (SELU) elementwise along x.
sigmoid(x, *[, constant])	Applies the sigmoid activation function.
softmax(x[, axis, constant])	Applies the softmax activation function.
soft_sign(x, *[, constant])	Returns the soft sign function x / (1 + |x|).
relu(x, *[, constant])	Applies the recitfied linear unit activation function.
tanh(x[, out, where, dtype, constant])	Hyperbolic tangent, element-wise.

mygrad.nnet.activations.elu

mygrad.nnet.activations.**elu**(*x: ArrayLike*, *alpha: Real*, *, *constant: Optional[bool] = None*) → Tensor

Returns the exponential linear activation (ELU) elementwise along x.

The ELU is given by *(exp(x) - 1) for x < 0 and x for x 0.*

Parameters

x
[ArrayLike] Input data.

alpha
[Real] The multiplicative factor on the negative activation.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor
The ELU function applied to *x* elementwise.

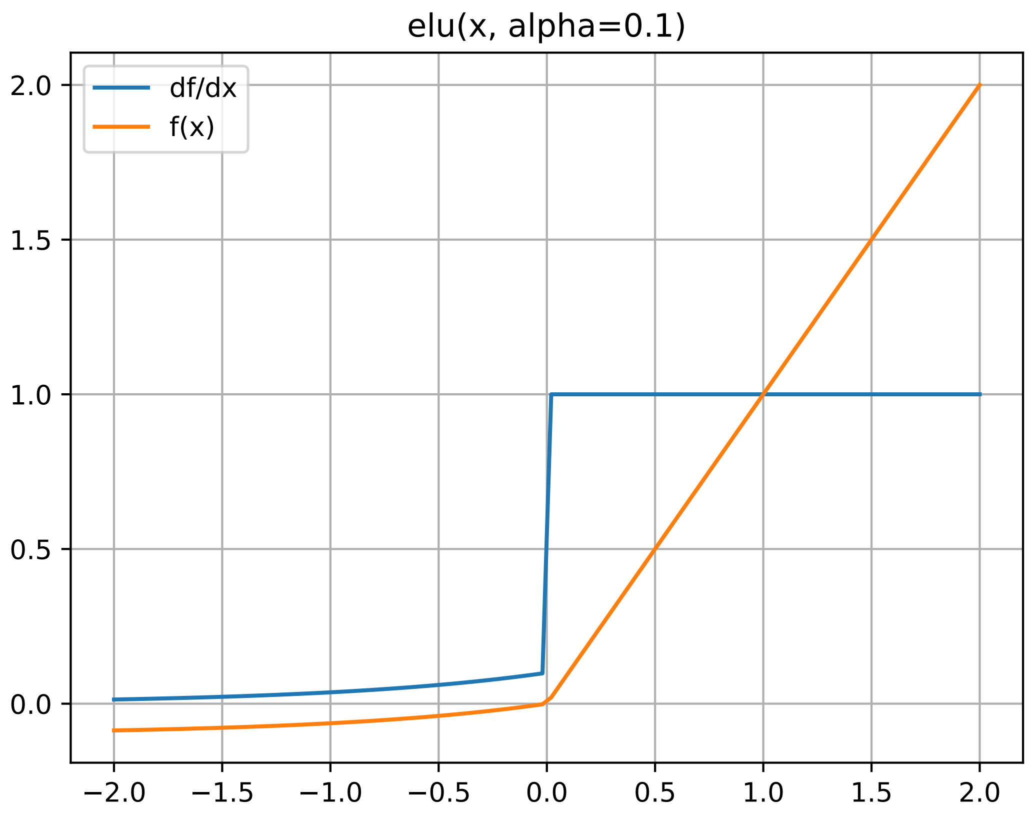
Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import elu
>>> x = mg.arange(-5, 6)
>>> x
Tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
>>> y = elu(x, alpha=0.1); y
Tensor([-0.09932621, -0.09816844, -0.09502129, -0.08646647, -0.06321206,
         0.        ,  1.        ,  2.        ,  3.        ,  4.        ,
         5.        ])
```

(continues on next page)

(continued from previous page)

```
>>> y.backward()
>>> x.grad
array([6.73794700e-04, 1.83156389e-03, 4.97870684e-03, 1.35335283e-02,
       3.67879441e-02, 1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
       1.00000000e+00, 1.00000000e+00, 1.00000000e+00])
```



mygrad.nnet.activations.glu

`mygrad.nnet.activations.glu`(*x: ArrayLike*, *axis: int = -1*, *, *constant: Optional[bool] = None*) → Tensor

Returns the Gated Linear Unit A * (B), where A and B are split from *x*.

Parameters

x
[ArrayLike] The input.

axis
[int, optional (default=-1)] The axis along which to split the input in half and apply the GLU.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient).

Returns

mygrad.Tensor
The result of applying the Gated Linear Unit elementwise to the input.

Notes

The Gated Linear Unit was proposed in the paper
"Language Modeling with Gated Convolutional Networks" Yann Dauphin, Angela Fan, Michael Auli, David Grangier

available at https://arxiv.org/abs/1612.08083

The GLU operation splits the input *x* in half along *axis*, storing the first half in A and the second in B. The return value is then A (B), where is elementwise multiplication and is the sigmoid function.

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import glu
>>> x = mg.arange(-5., 5.)
>>> x
Tensor([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
>>> y = glu(x); y
Tensor([-2.5       , -2.92423431, -2.64239123, -1.90514825, -0.98201379])
>>> y.backward()
>>> x.grad
array([ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0])
```

mygrad.nnet.activations.hard_tanh

`mygrad.nnet.activations.`**`hard_tanh`**(*x: ArrayLike*, ***, *lower_bound: Real = -1*, *upper_bound: Real = 1*, *constant: Optional[bool] = None*) → Tensor

Returns the hard hyperbolic tangent function.

The hard_tanh function is *lower_bound* where *x* <= *lower_bound*, *upper_bound* where *x* >= *upper_bound*, and *x* where *lower_bound* < *x* < *upper_bound*.

Parameters

x
[ArrayLike] The input, to which to apply the hard tanh function.

lower_bound
[Real, optional (default=-1)] The lower bound on the hard tanh.

upper_bound
[Real, optional (default=1)] The upper bound on the hard tanh.

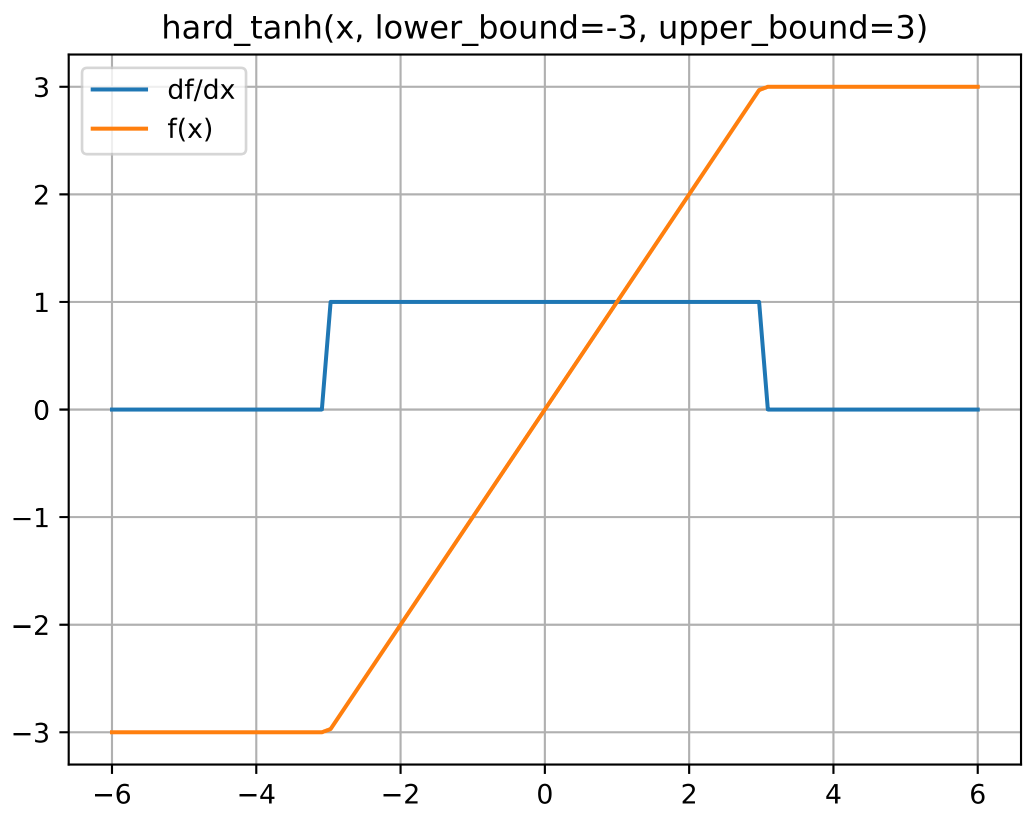
constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient).

Returns

mygrad.Tensor
The result of applying the "hard-tanh" function elementwise to *x*.

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import hard_tanh
>>> x = mg.arange(-5, 6)
>>> x
Tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
>>> y = hard_tanh(x, lower_bound=-3, upper_bound=3); y
Tensor([-3, -3, -3, -2, -1,  0,  1,  2,  3,  3,  3])
>>> y.backward()
>>> x.grad
array([0., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0.])
```



mygrad.nnet.activations.leaky_relu

`mygrad.nnet.activations.leaky_relu`(*x: ArrayLike*, *slope: float*, ***, *constant: Optional[bool] = None*) → Tensor

Returns the leaky rectified linear activation elementwise along x.

The leaky ReLU is given by *max(x, 0) + slope*min(x, 0)*.

Parameters

x
[ArrayLike] Input data.

slope
[Union[Real, mygrad.Tensor]] The slope of the negative activation.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient).

Returns

mygrad.Tensor
The result of apply the "leaky relu" function elementwise to *x*.

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import leaky_relu
>>> x = mg.arange(-5, 6)
>>> x
Tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
>>> y = leaky_relu(x, slope=0.1); y
>>> Tensor([-0.5, -0.4, -0.3, -0.2, -0.1,  0. ,  1. ,  2. ,  3. ,  4. ,  5. ])
>>> y.backward()
>>> x.grad
array([0.1, 0.1, 0.1, 0.1, 0.1, 0. , 1. , 1. , 1. , 1. , 1. ])
```

mygrad.nnet.activations.logsoftmax

`mygrad.nnet.activations.logsoftmax`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = -1*, ***, *constant: Optional[bool] = None*) → Tensor

Applies the log-softmax activation function:

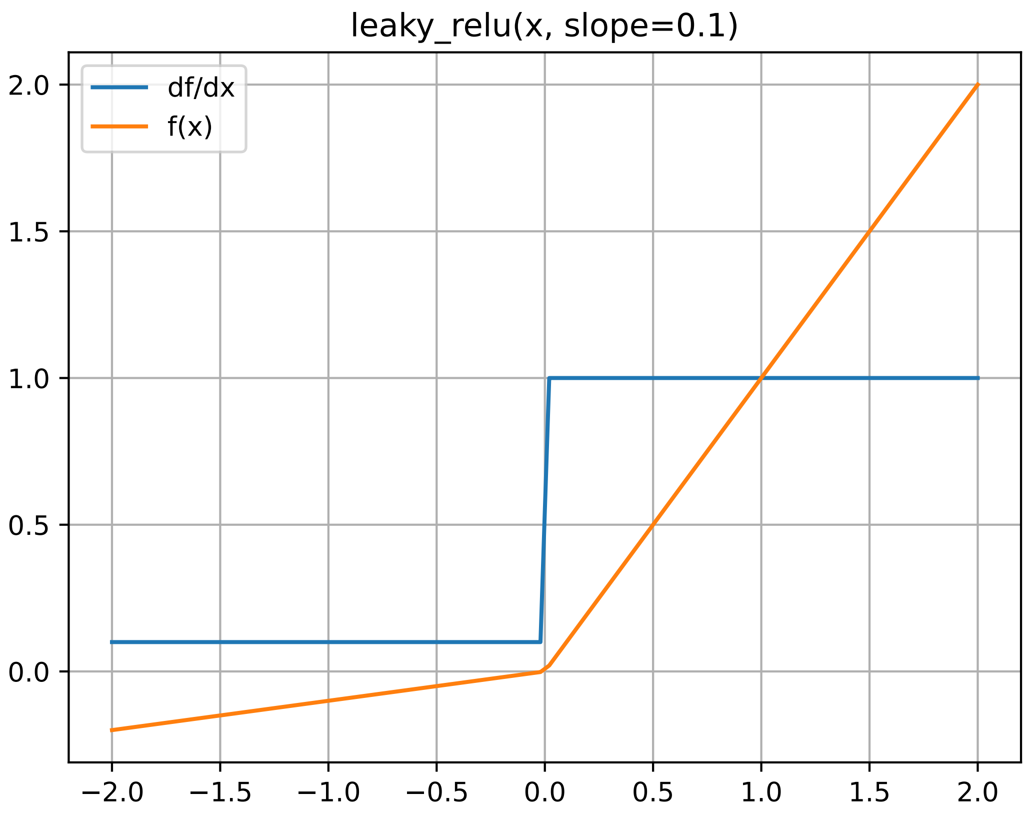
```
f(x) = log ( exp(x) / sum( exp(x) ) )
```

Computes the log-softmax over one or more axes of an ND-tensor.

Parameters

x
[ArrayLike]

axis
[Union[None, int, Tuple[int, ...]], optional (default=-1)] The axis/axes over which to compute the log-softmax. By default, the log-softmax is computed over the trailing axis.


leaky_relu(x, slope=0.1)
df/dx
f(x)
2.0
1.5
1.0
0.5
0.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

constant
[constant][Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

log_softmax
[mygrad.Tensor] Tensor with same shape as `x`

Notes

- N is the number of samples in the batch.
- C is the number of possible classes for which scores are provided.

This implements a numerically-stable version of log-softmax, compared to the naive implementation using `mygrad.log`, `mygrad.exp`, and `mygrad.sum`.

Given the shape-(N, C) tensor of scores, `x`, the softmax classification probabilities are computed. That is, the score for class-k of a given datum (s_k) is normalized using the 'softmax' transformation:

$$p_k = \log \frac{e^{s_k}}{\sum_{i=1}^{C} e^{s_i}}$$

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import logsoftmax
>>> x = mg.Tensor([[  2.,   2.,    2.],
...                 [2E50, 2E50,  1E50]])
>>> logsoftmax(x)
Tensor([[-1.09861229e+00, -1.09861229e+00, -1.09861229e+00],
        [ 0.00000000e+00,  0.00000000e+00, -1.00000000e+50]])
```

mygrad.nnet.activations.selu

`mygrad.nnet.activations.selu`(*x: ArrayLike*, *, *constant: Optional[bool] = None*) → Tensor

Returns the scaled exponential linear activation (SELU) elementwise along x.

The SELU is given by (exp(x) - 1) for x < 0 and x for x 0.

Parameters

x
[ArrayLike] Input data.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

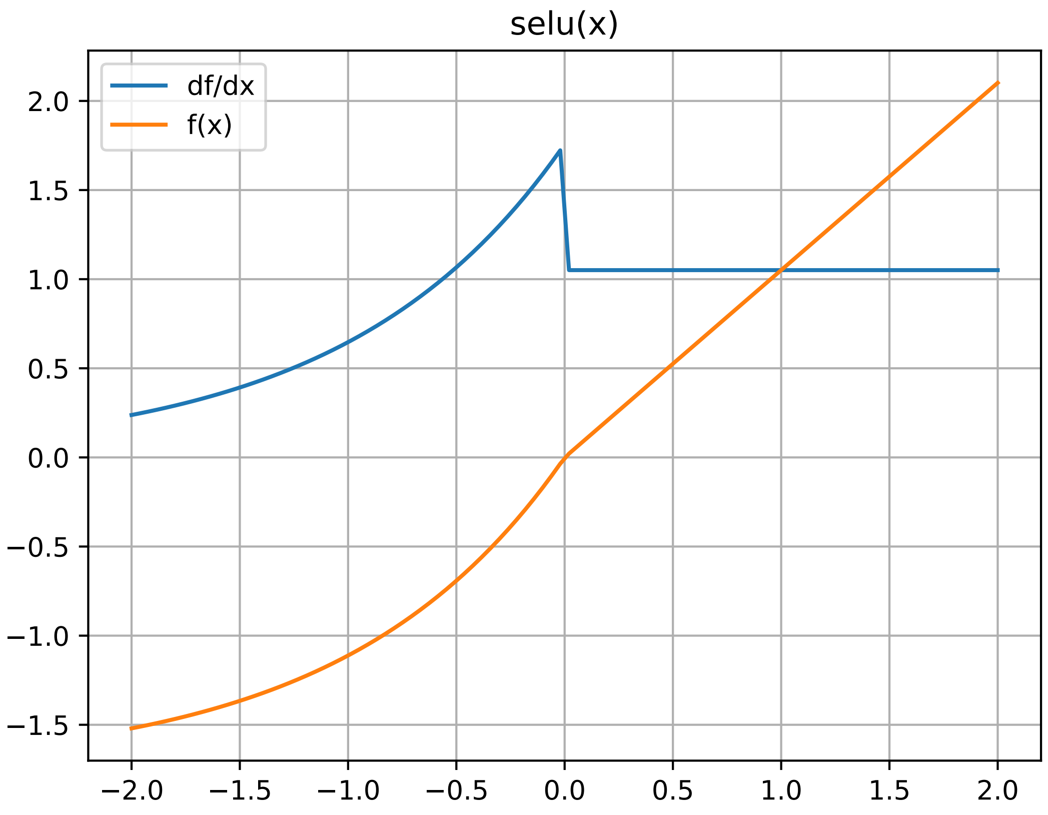
mygrad.Tensor
The SELU function applied to *x* elementwise.

References

[1]

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import selu
>>> x = mg.arange(-5, 6)
>>> x
Tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
>>> y = selu(x, alpha=0.1); y
Tensor([-1.74625336, -1.72589863, -1.67056873, -1.52016647, -1.11133074,
        0.        ,  1.05070099,  2.10140197,  3.15210296,  4.20280395,
        5.25350494])
```



mygrad.nnet.activations.sigmoid

`mygrad.nnet.activations.sigmoid`(*x: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

Applies the sigmoid activation function:

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

Parameters

x
[ArrayLike] sigmoid is applied element-wise on `x`.

constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import sigmoid
>>> x = mg.linspace(-5, 5, 10)
>>> sigmoid(x)
Tensor([0.00669285, 0.02005754, 0.0585369 , 0.1588691 , 0.36457644,
    0.63542356, 0.8411309 , 0.9414631 , 0.97994246, 0.99330715])
```

mygrad.nnet.activations.softmax

`mygrad.nnet.activations.softmax`(*x: ArrayLike*, *axis: Union[None, int, Tuple[int, ...]] = -1*, ***, *constant: Optional[bool] = None*) → Tensor

Applies the softmax activation function:

```
f(x) = exp(x) / sum( exp(x) )
```

Computes the softmax over one or more axes of an ND-tensor.

Parameters

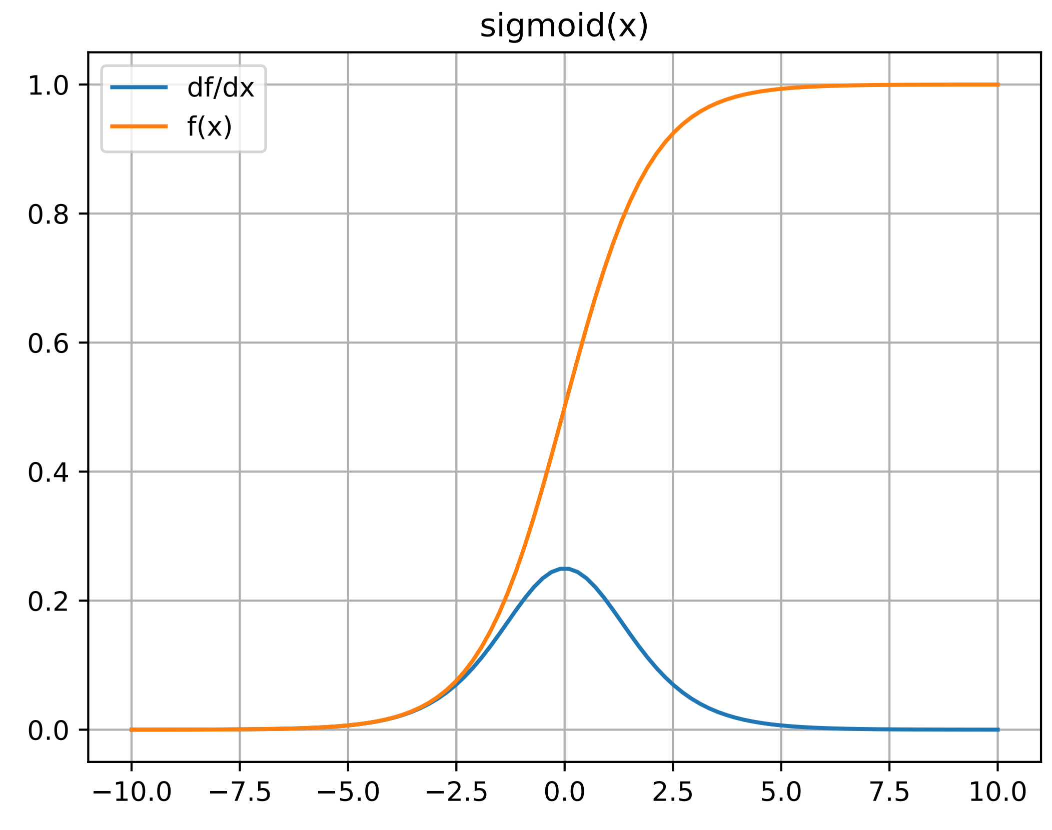
x
[array_like]

axis
[Union[None, int, Tuple[int, ...]], optional (default=-1)] The axis/axes over which to compute the softmax. By default, the softmax is computed over the trailing axis.

constant
[bool, optional(default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor


sigmoid(x)
df/dx
f(x)
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Notes

- N is the number of samples in the batch.
- C is the number of possible classes for which scores are provided.

This implements a numerically-stable version of softmax, however log-softmax is still the more numerically stable activation function.

Given the shape-(N, C) tensor of scores, `x`, the softmax classification probabilities are computed. That is, the score for class-k of a given datum (s_k) is normalized using the 'softmax' transformation:

$$p_k = \frac{e^{s_k}}{\sum_{i=1}^{C} e^{s_i}}$$

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import softmax
>>> x = mg.Tensor([[ 2.,  2.,  2.],
...                [2E50, 2E50,  1E50]])
>>> softmax(x)
Tensor([[0.33333333, 0.33333333, 0.33333333],
        [0.5       , 0.5       , 0.        ]])
```

mygrad.nnet.activations.soft_sign

`mygrad.nnet.activations.soft_sign`(*x: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

Returns the soft sign function x / (1 + |x|).

Parameters

x
[ArrayLike] Input data.

constant
[boolean, optional (default=False)] If `True`, the returned tensor is a constant (it does not back-propagate a gradient).

Returns

mygrad.Tensor
The soft sign function applied to *x* elementwise.

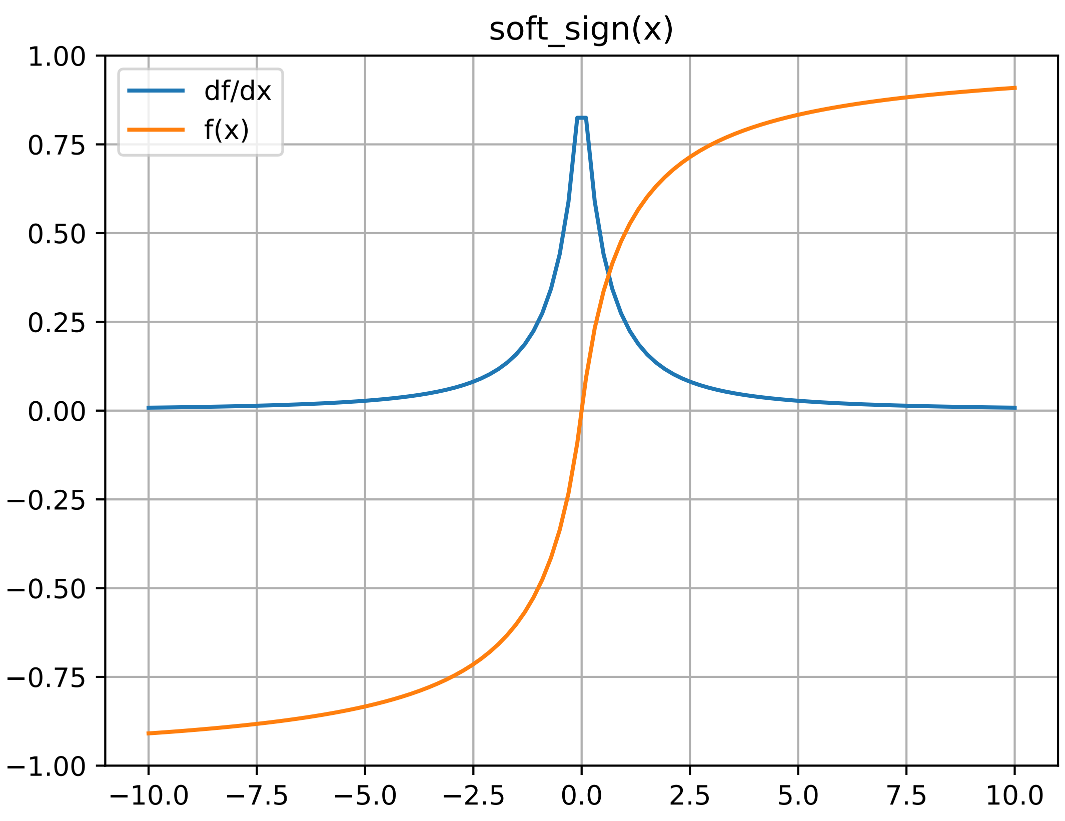
Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet.activations import soft_sign
>>> x = mg.arange(-5, 6)
>>> x
Tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
>>> y = soft_sign(x); y
Tensor([-0.83333333, -0.8       , -0.75      , -0.66666667, -0.5       ,
```

(continues on next page)

(continued from previous page)

```
        0.        ,  0.5       ,  0.66666667,  0.75      ,  0.8       ,
        0.83333333])
```



mygrad.nnet.activations.relu

mygrad.nnet.activations.**relu**(*x: ArrayLike*, ***, *constant: Optional[bool] = None*) → Tensor

Applies the recitified linear unit activation function:

```
f(x) = {x, x > 0
        0, x <= 0 }
```

Parameters

x
[ArrayLike] relu is applied element-wise on `x`.

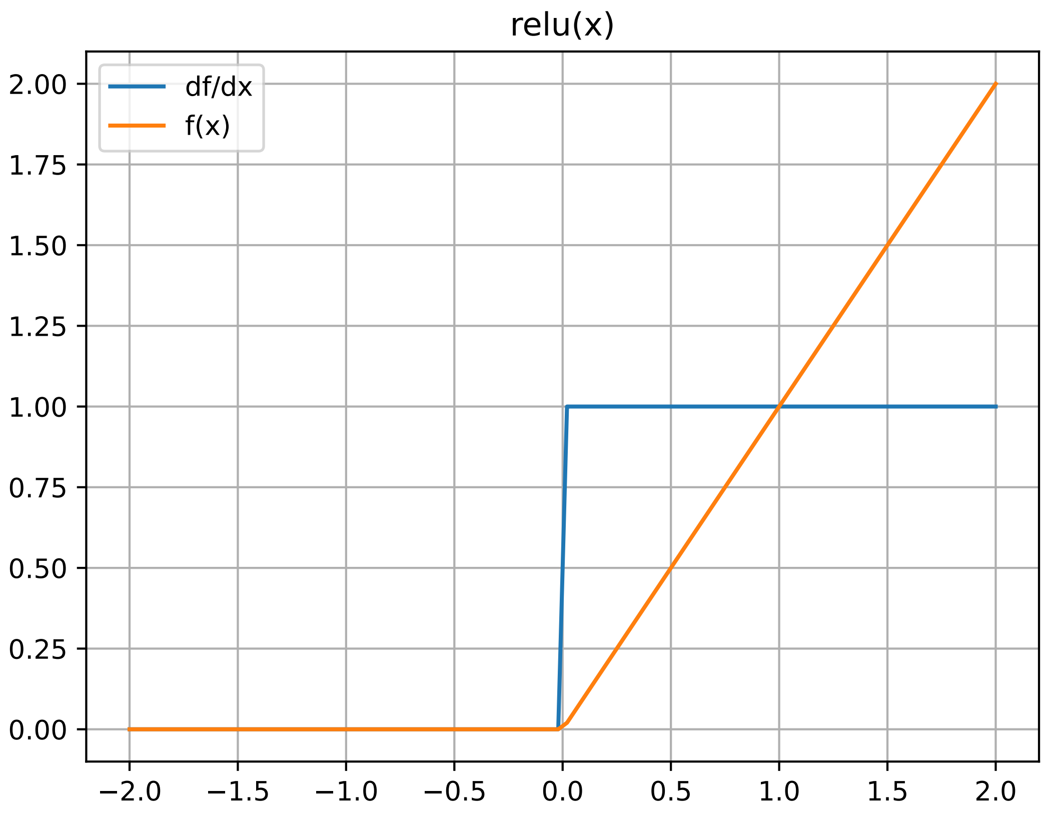
constant
[Optional[bool]] If `True`, the returned tensor is a constant (it does not back-propagate a gradient)

Returns

mygrad.Tensor

Examples

```
>>> import mygrad as mg
>>> from mygrad.nnet import relu
>>> x = mg.linspace(-5, 5, 5)
>>> x
Tensor([-5. , -2.5,  0. ,  2.5,  5. ])
>>> relu(x)
Tensor([-0. , -0. ,  0. ,  2.5,  5. ])
>>> relu(x).backward()
>>> x.grad  # d(relu(x))/dx
array([0., 0., 0., 1., 1.])
```



mygrad.nnet.activations.tanh

class `mygrad.nnet.activations.tanh`(*x: ArrayLike*, *out: Optional[Union[Tensor, ndarray]] = None*, ***, *where: Mask = True*, *dtype: DTypeLikeReals = None*, *constant: Optional[bool] = None*)

Hyperbolic tangent, element-wise.

This docstring was adapted from that of numpy.tanh [3]

Parameters

x
[ArrayLike] Input tensor.

out
[Optional[Union[Tensor, ndarray]]] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated tensor is returned.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

where
[Mask] This condition is broadcast over the input. At locations where the condition is True, the `out` tensor will be set to the ufunc result. Elsewhere, the `out` tensor will retain its original value. Note that if an uninitialized *out* tensor is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

dtype
[Optional[DTypeLikeReals]] The dtype of the resulting tensor.

Returns

y
[Tensor] The corresponding hyperbolic tangent values.

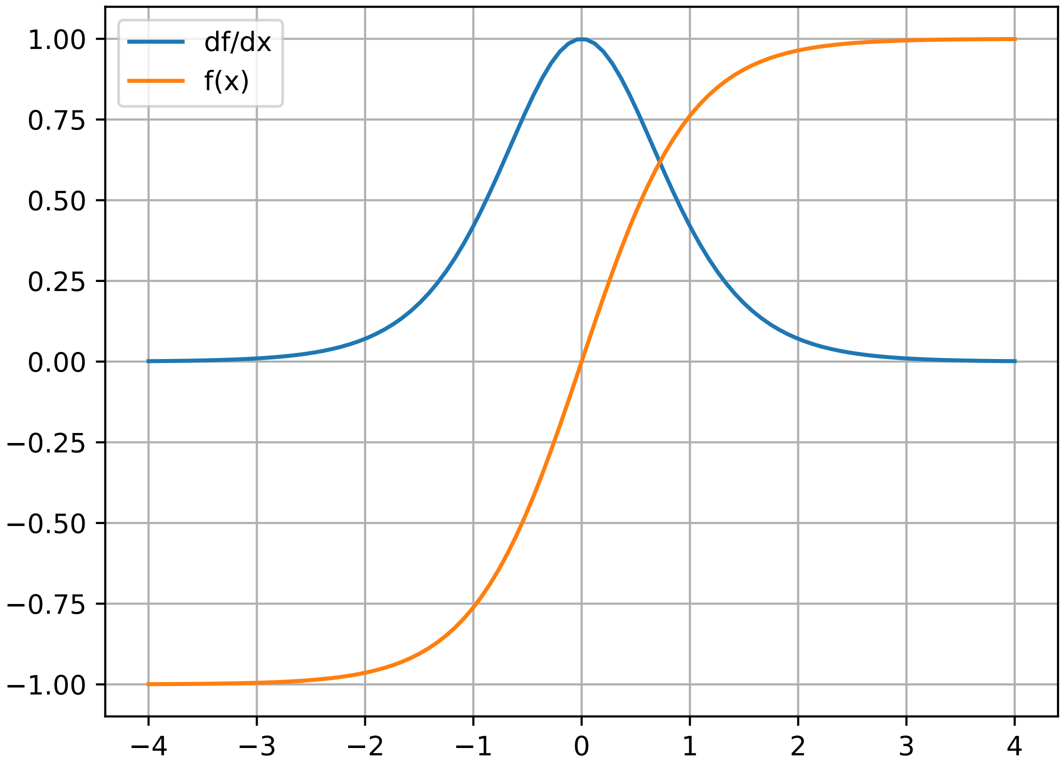
References

[1], [2], [3]

Examples

```
>>> import mygrad as mg
>>> x = mg.linspace(-2, 2, 10)
>>> y = mg.tanh(x); y
Tensor([-0.96402758, -0.9146975 , -0.8044548 , -0.58278295, -0.21863508,
     0.21863508,  0.58278295,  0.8044548 ,  0.9146975 ,  0.96402758])
```

```
>>> y.backward()  # compute d(tanh)/dx
>>> x.grad
array([0.07065082, 0.16332849, 0.35285247, 0.66036404, 0.9521987 ,
       0.9521987 , 0.66036404, 0.35285247, 0.16332849, 0.07065082])
```


df/dx
f(x)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−4
−3
−2
−1
0
1
2
3
4

Attributes

identity
signature

Methods

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

`accumulate`([axis, dtype, out, constant])	Not implemented
`at`(indices[, b, constant])	Not implemented
`outer`(b, *[, dtype, out])	Not Implemented
`reduce`([axis, dtype, out, keepdims, ...])	Not Implemented
`reduceat`(indices[, axis, dtype, out])	Not Implemented

Attributes

`identity`
`nargs`
`nin`
`nout`
`ntypes`
`signature`
`types`

3.12.4 Initializers

dirac(*shape[, dtype, constant])	Initialize a `mygrad.Tensor` according to the Dirac initialization procedure described by Zagoruyko and Komodakis.
glorot_normal(*shape[, gain, dtype, constant])	Initialize a `mygrad.Tensor` according to the normal initialization procedure described by Glorot and Bengio.
glorot_uniform(*shape[, gain, dtype, constant])	Initialize a `mygrad.Tensor` according to the uniform initialization procedure described by Glorot and Bengio.
he_normal(*shape[, gain, dtype, constant])	Initialize a `mygrad.Tensor` according to the normal initialization procedure described by He et al.
he_uniform(*shape[, gain, dtype, constant])	Initialize a `mygrad.Tensor` according to the uniform initialization procedure described by He et al.
normal(*shape[, mean, std, dtype, constant])	Initialize a `mygrad.Tensor` by drawing from a normal (Gaussian) distribution.
uniform(*shape[, lower_bound, upper_bound, ...])	Initialize a `mygrad.Tensor` by drawing from a uniform distribution.

mygrad.nnet.initializers.dirac

`mygrad.nnet.initializers.`**`dirac`**(**shape: int*, *dtype=<class 'numpy.float32'>*, *constant: ~typing.Optional[bool] = None*) → Tensor

Initialize a `mygrad.Tensor` according to the Dirac initialization procedure described by Zagoruyko and Komodakis.

Parameters

shape
[Sequence[int]] The shape of the output Tensor. Note that `shape` must be at least two-dimensional.

dtype
[data-type, optional (default=float32)] The data type of the output tensor.

constant
[Optional[bool]] If `True`, this tensor is treated as a constant, and thus does not facilitate back propagation (i.e. `constant.grad` will always return `None`).

Defaults to `False` for float-type data. Defaults to `True` for integer-type data.

Integer-type tensors must be constant.

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values initialized according to the Dirac initialization.

mygrad.nnet.initializers.glorot_normal

`mygrad.nnet.initializers.`**`glorot_normal`**`(`**shape*, *gain=1*, *dtype=<class 'numpy.float32'>*, *constant=None*`)`

Initialize a `mygrad.Tensor` according to the normal initialization procedure described by Glorot and Bengio.

Parameters

shape
[Sequence[int]] The shape of the output Tensor. Note that `shape` must be at least two-dimensional.

gain
[Real, optional (default=1)] The gain (scaling factor) to apply.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If `True`, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values initialized according to the glorot normal initialization.

Notes

Glorot and Bengio put forward this initialization in the paper
"Understanding the Difficulty of Training Deep Feedforward Neural Networks"

http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

A Tensor W initialized in this way should be drawn from a distribution about

$$\mathcal{N}(0, \frac{\sqrt{2}}{\sqrt{n_j + n_{j+1}}})$$

mygrad.nnet.initializers.glorot_uniform

`mygrad.nnet.initializers.`**`glorot_uniform`**`(`**shape*, *gain=1*, *dtype=<class 'numpy.float32'>*, *constant=None*`)`

Initialize a `mygrad.Tensor` according to the uniform initialization procedure described by Glorot and Bengio.

Parameters

shape
[Sequence[int]] The shape of the output Tensor. Note that `shape` must be at least two-dimensional.

gain
[Real, optional (default=1)] The gain (scaling factor) to apply.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If `True`, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values initialized according to the glorot uniform initialization.

Notes

Glorot and Bengio put forward this initialization in the paper
"Understanding the Difficulty of Training Deep Feedforward Neural Networks"

http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

A Tensor W initialized in this way should be drawn from a distribution about

$$U[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$$

mygrad.nnet.initializers.he_normal

`mygrad.nnet.initializers.`**`he_normal`**`(`**shape*, *gain=1*, *dtype=<class 'numpy.float32'>*, *constant=None*`)`

Initialize a `mygrad.Tensor` according to the normal initialization procedure described by He et al.

Parameters

shape
[Sequence[int]] The shape of the output Tensor. Note that `shape` must be at least two-dimensional.

gain
[Real, optional (default=1)] The gain (scaling factor) to apply.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If `True`, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values initialized according to the He normal initialization.

Notes

He, Zhang, Ren, and Sun put forward this initialization in the paper
“Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”

https://arxiv.org/abs/1502.01852

A Tensor W initialized in this way should be drawn from a distribution about

$$\mathcal{N}(0, \sqrt{\frac{2}{(1+a^2)n_l}})$$

where a is the slope of the rectifier following this layer, which is incorporated using the *gain* variable above.

The guidance put forward in that paper is that this initialization procedure should be preferred over the `mygrad.nnet.initializers.glorot_*` functions especially when rectifiers (e.g. ReLU, PReLU, leaky_relu) in very deep (> 1-20 or so layer) networks.

Examples

```
>>> from mygrad.nnet.initializers import he_normal
>>> he_normal(2, 3)
Tensor([[-2.3194842 ,  0.45956254, -0.28709933],
        [-0.15776408,  0.6777564 , -0.05587448]], dtype=float32)
```

```
>>> he_normal(4, 2, gain=5/3, dtype="float64", constant=True)
Tensor([[ 0.25962918,  1.1503933 ],
        [-0.13638746,  0.10581096],
        [ 1.44805926,  0.51367645],
        [-0.32018705, -0.80306442]])
```

```
>>> he_normal(2, 1, 2, dtype="float16")
Tensor([[[ 0.8057 , -0.2922 ]],
        [[ 0.12213, -0.715  ]]], dtype=float16)
```

mygrad.nnet.initializers.he_uniform

mygrad.nnet.initializers.**he_uniform**(**shape*, *gain=1*, *dtype=<class 'numpy.float32'>*, *constant=None*)

Initialize a `mygrad.Tensor` according to the uniform initialization procedure described by He et al.

Parameters

shape
[Sequence[int]] The shape of the output Tensor. Note that `shape` must be at least two-dimensional.

gain
[Real, optional (default=1)] The gain (scaling factor) to apply.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If *True*, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values initialized according to the He uniform initialization.

Notes

He, Zhang, Ren, and Sun put forward this initialization in the paper
"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"

https://arxiv.org/abs/1502.01852

A Tensor W initialized in this way should be drawn from a distribution about

$$U[-\sqrt{\frac{6}{(1+a^2)n_l}}, \sqrt{\frac{6}{(1+a^2)n_l}}]$$

where a is the slope of the rectifier following this layer, which is incorporated using the *gain* variable above.

The guidance put forward in that paper is that this initialization procedure should be preferred over the `mygrad.nnet.initializers.glorot_*` functions especially when rectifiers (e.g. ReLU, PReLU, leaky_relu) in very deep (> 1-20 or so layer) networks.

Examples

```
>>> from mygrad.nnet.initializers import he_uniform
>>> he_uniform(2, 3)
Tensor([[-0.97671795,  0.85518736, -0.8187388 ],
        [ 0.7599437 ,  0.94951814, -0.96755147]], dtype=float32)
```

```
>>> he_uniform(4, 2, gain=5/3, dtype="float64", constant=True)
Tensor([[-1.10372799, -0.16472136],
        [-1.32614867,  1.14142637],
        [ 0.78044471,  0.20562334],
        [-1.23968259,  1.0057054 ]])
```

```
>>> he_uniform(2, 1, 2, dtype="float16")
Tensor([[[-0.1233,  0.1023]],
        [[ 0.3845,  0.1003]]], dtype=float16)
```

mygrad.nnet.initializers.normal

`mygrad.nnet.initializers.`**`normal`**(**shape*, *mean=0*, *std=1*, *dtype=<class 'numpy.float32'>*, *constant=None*)

Initialize a `mygrad.Tensor` by drawing from a normal (Gaussian) distribution.

Parameters

shape
[Sequence[int]] The output shape.

mean
[Real, optional (default=0)] The mean of the distribution from which to draw.

std
[Real, optional (default=1)] The standard deviation of the distribution from which to draw.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If True, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values drawn from (, ^{2}), where =``mean`` and =``std``.

Examples

```
>>> from mygrad.nnet.initializers import normal
>>> normal(1, 2, 3)
Tensor([[[-0.06481607, -0.550582  ,  0.04689528],
         [ 0.82973075,  2.83742   ,  1.0964519 ]]], dtype=float32)
```

```
>>> normal(2, 2, dtype="float16", constant=True)
Tensor([[-1.335 ,  0.9297],
        [ 1.746 , -0.1222]], dtype=float16)
```

```
>>> normal(5, dtype="float64")
Tensor([-0.03875407,  0.65368466, -0.72636993,  1.57404148, -1.17444345])
```

mygrad.nnet.initializers.uniform

`mygrad.nnet.initializers.uniform`(**shape*, *lower_bound=0*, *upper_bound=1*, *dtype=<class 'numpy.float32'>*, *constant=None*)

Initialize a `mygrad.Tensor` by drawing from a uniform distribution.

Parameters

shape
[Sequence[int]] The output shape.

lower_bound
[Real, optional (default=0)] Lower bound on the output interval, inclusive.

upper_bound
[Real, optional (default=1)] Upper bound on the output interval, exclusive.

dtype
[data-type, optional (default=float32)] The data type of the output tensor; must be a floating-point type.

constant
[bool, optional (default=False)]

If True, the returned tensor is a constant (it
does not back-propagate a gradient).

Returns

mygrad.Tensor, shape=``shape``
A Tensor, with values drawn uniformly from [lower_bound, upper_bound).

Examples

```
>>> from mygrad.nnet.initializers import uniform
>>> uniform(2, 3)
Tensor([[0.8731087 , 0.30872548, 0.75528544],
        [0.55404514, 0.7652222 , 0.4955769 ]], dtype=float32)
```

```
>>> uniform(2, 2, lower_bound=-1, upper_bound=3)
Tensor([[ 1.9151938 , -0.28968155],
        [-0.01240687, -0.24448799]], dtype=float32)
```

```
>>> uniform(5, dtype="float16", constant=True)
Tensor([0.5186, 0.1481, 0.3745, 0.941 , 0.331 ], dtype=float16)
```

3.12.5 Sliding Window View Utility

sliding_window_view(arr, window_shape, step)	Create a sliding window view over the trailing dimensions of an array.

mygrad.sliding_window_view

mygrad.**sliding_window_view**(*arr*, *window_shape*, *step*, *dilation=None*)

Create a sliding window view over the trailing dimensions of an array. No copy is made unless the input array is not contiguous in memory.

The window is applied only to valid regions of `arr`, but is applied greedily.

See Notes section for details.

Parameters

arr
[numpy.ndarray, shape=(..., [x, (...), z])] C-contiguous array over which sliding view-window is applied along the trailing dimensions `[x, ..., z]`, as determined by the length of `window_shape`.

If `arr` is not C-contiguous, it will be replaced by `numpy.ascontiguousarray(arr)`

window_shape
[Sequence[int]] Specifies the shape of the view-window: `[Wx, (...), Wz]`. The length of *window_shape* determines the length of `[x, (...) , z]`.

step
: [Union[int, Sequence[int]]] The step sized used along the `[x, (...), z]` dimensions: `[Sx, (...), Sz]`. If a single integer is specified, a uniform step size is used.

dilation
: [Optional[Union[int, Sequence[int]]]] The dilation factor used along the `[x, (...), z]` directions: `[Dx, (...), Dz]`. If no value is specified, a dilation factor of 1 is used along each direction. Dilation specifies the step size used when filling the window's elements

Returns

numpy.ndarray
: A contiguous view of `arr`, of shape `([X, (...), Z], ..., [Wx, (...), Wz])`, where `[X, ..., Z]` is the shape of the grid on which the window was applied. See Notes sections for more details.

Raises

ValueError, TypeError
: Invalid step-size, window shape, or dilation

Notes

Window placement:

Given a dimension of size x, with a window of size W along this dimension, applied with stride S and dilation D, the window will be applied:

```
X = (x - (W - 1) * D + 1) // S + 1
```

number of times along that dimension.

Interpreting output:

In general, given an array `arr` of shape (..., x, (...), z), and:

```
out = sliding_window_view(arr, window_shape=[Wx, (...), Wz], step=[Sx, (...),␣
↪Sz])
```

then indexing `out` with `[ix, (...), iz]` produces the following view of `x`:

```
out[ix, (...), iz] ==
    x[..., ix*Sx:(ix*Sx + Wx*Dx):Dx, (...), iz*Sz:(iz*Sz + Wz*Dz):Dz]
```

For example, suppose `arr` is an array of shape-(10, 12, 6). Specifying sliding window of shape `(3, 3)` with step size `(2, 2)`, dilation `(2, 1)` will create the view:

```
[[arr[:,  0:6:2, 0:3], arr[:,   0:6:3, 3:6]]
 [arr[:, 6:12:2, 0:3], arr[:, 6:12:12, 3:6]]]
```

producing a view of shape `(2, 2, 10, 3, 3)` in total.

Examples

```
>>> import numpy as np
>>> x = np.arange(36).reshape(6, 6)
>>> x
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

Apply an 3x2 window with step-sizes of (2, 2). This results in the window being placed twice along axis-0 and three times along axis-1.

```
>>> y = sliding_window_view(x, step=(2, 2), window_shape=(3, 2))
>>> y.shape
(2, 3, 3, 2)
```

window applied at (0, 0)

```
>>> y[0, 0]
array([[ 0,  1],
       [ 6,  7],
       [12, 13]])
```

window applied at (2, 0)

```
>>> y[1, 0]
array([[12, 13],
       [18, 19],
       [24, 25]])
```

window applied at (0, 2)

```
>>> y[0, 1]
array([[ 2,  3],
       [ 8,  9],
       [14, 15]])
```

verify that an element in this window-view is correct

```
>>> i, j = np.random.randint(0, 2, size=2)
>>> wx, wy = (2, 2)
>>> sx, sy = (2, 2)
>>> np.all(y[i, j] == x[..., i*sx:(i*sx + wx), j*sy:(j*sy + wy)])
True
```

3.13 Input and Output

3.13.1 NumPy binary files (NPY, NPZ)

save(file, tensor)	Saves a tensor and its gradient information.
load(file)	Loads a saved Tensor and its gradient information (if applicable).

mygrad.save

mygrad.**save**(*file: Union[str, Path, BinaryIO], tensor: Tensor*) → None

Saves a tensor and its gradient information.

This docstring was adapted from that of numpy.save()

Parameters

file
[str | Path | BinaryIO] The file or file-path that where the tensor data and its gradient will be saved. Note that the file will be saved as a .npz file.

tensor
[Tensor] The tensor to be saved. If it has an associated gradient, that will be saved as well.

See also:

mygrad.load

Notes

This function uses `numpy.savez(file, data=tensor.data, grad=tensor.grad)` to save the tensor's data and its gradient. No `grad` field is included if the tensor does not have a gradient.

Examples

```
>>> import mygrad as mg
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = mg.arange(10.0)
>>> mg.save(outfile, x)
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> mg.load(outfile)
Tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

An example of saving a tensor that has an associated gradient.

```
>>> (x * x).backward()
>>> x.grad
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
>>> outfile = TemporaryFile()
>>> x = mg.arange(10.0)
```

(continues on next page)

(continued from previous page)

```
>>> mg.save(outfile, x)
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> loaded = mg.load(outfile)
>>> loaded
Tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> loaded.grad
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
```

mygrad.load

mygrad.**load**(*file: Union[str, Path, BinaryIO]*) → Tensor

Loads a saved Tensor and its gradient information (if applicable).

This docstring was adapted from that of numpy.load()

Parameters

file
[str | Path | BinaryIO] The name of the file that holds the tensor data to load.

Returns

loaded
[Tensor] The loaded tensor (whose gradient will be loaded if it was saved).

See also:

mygrad.save

Examples

```
>>> import mygrad as mg
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = mg.arange(10.0)
>>> mg.save(outfile, x)
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> mg.load(outfile)
Tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

An example of saving a tensor that has an associated gradient.

```
>>> (x * x).backward()
>>> x.grad
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
>>> outfile = TemporaryFile()
>>> x = mg.arange(10.0)
>>> mg.save(outfile, x)
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> loaded = mg.load(outfile)
>>> loaded
Tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

(continues on next page)

(continued from previous page)

```
>>> loaded.grad
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
```

3.14 Computational graph visualization(mygrad.computational_graph)

build_graph(fin[, names, render, save, ...])	Builds and renders a computational graph.

3.14.1 mygrad.computational_graph.build_graph

mygrad.computational_graph.**build_graph**(*fin*, *names=None*, *, *render=True*, *save=False*, *dims=False*, *dtypes=False*, *sum_stats=False*)

Builds and renders a computational graph.

Parameters

fin
[mygrad.Tensor] The tensor object that will be the final node in the computational graph.

names
[Optional[Dict[str, Union[mygrad.Tensor, numpy.ndarray]]]] A dictionary that maps names of Tensors to Tensor objects. If an argument is passed to names, the key name that maps to a Tensor included in the computational graph will be used as a label for the Tensor's node. If no argument is passed, the nodes on the computational graph will display the full Tensor.

To use the names assigned in the local environment, pass `names=locals()` to the build_graph function.

If different names are used from the local environment, the key must map to the exact Tensor object. A new Tensor or copy of the original Tensor should not be created as the value in the dictionary.

Only instances of mygrad.Tensor or numpy.ndarray can have labels assigned to Nodes. If a list or tuple is used in an operation with a Tensor, and names is not None, the Node label will be set to *Constant*. If a list or tuple is used in multiple operations, a unique Node will be created for each time it is used.

A scalar will always be used as the label for a 0-dimensional Tensor's Node.

render
[bool, optional (default=True)] If True, build_graph will return a graphviz Digraph object that, when called, will render the computational graph in a Jupyter notebook or the Jupyter Qt console. If False, nothing is returned.

save
[bool, optional (default=False)] If True, build_graph will save a rendered computational graph to the current working directory as `computational_graph.pdf`.

dims
[bool, optional (default=False)] If True, Tensor dimensions are added to Node labels. Dimensions will not be displayed for scalar values.

dtypes
[bool, optional (default=False)] If True, Tensor data types are added to Node labels.

sum_stats
[bool, optional (default=False)] If True, Tensor minimums, maximums, medians, and means are added to Node labels. These will not be displayed for scalar values.

Returns

Union[graphviz.Digraph, None]

Notes

build_graph requires that Graphviz is installed.

3.15 Changelog

This is a record of all past mygrad releases and what went into them, in reverse chronological order. All previous releases should still be available on pip.

3.15.1 2.2.0 - 2023-01-03

- MyGrad is now tested against Python 3.11. (pull request #411)
- `mygrad.bool8` has been removed. Use `mygrad.bool_` instead. (pull request #411)
- Adds ufunc support for `resolve_dtypes`. (pull request #411)
- Modifies automatic differentiation framework to be simpler and more memory efficient. In the future, MyGrad will be able to expose an API akin to torch.autograd.grad. (pull request #407)
- MyGrad's CI now enforces formatting and spell check requirements on all pull requests. (pull request #411)

3.15.2 2.1.0 - 2022-01-01

New Functions and Utilities

The following differentiable functions are now supported by MyGrad, and "drop-in" overrides for their NumPy counterparts are supported as well.

- *atleast_1d()*
- *atleast_2d()*
- *atleast_3d()*

Basic tensor save/load functionality has been added (thanks to @kw-0).

- *save()*
- *load()*

Improvements

- *clip()* and `Tensor.clip` now accept an `out` target, permitting in-place operations.
- The method `Tensor.__index__()` is now implemented, which permits scalar integer-valued tensors to be used to index into Python sequences.
- Added Python 3.10 to our automated test matrix.

Compatibility-Breaking Changes

- In accordance with NEP 29 we are dropping support for NumPy versions below 1.19. However, MyGrad will not drop support for Python 3.7; to remain as lightweight and flexible as possible we will support minor versions of Python up until their EOL or until our minimal NumPy dependency drops support – whichever occurs first.
- The interface to *arange()* was changed from `arange(start, stop=None, step=None, ...)` to `arange([start,] stop[, step,], ...)`. This provides exact parity with NumPy's arange function.
- The derivatives of *absolute()* and *norm()* have been revised such that in cases where the derivatives used to be `nan`, those entries will now be `0`. Both functions can now be passed `nan_to_num=False` to enable the previous, more rigorous behavior. See PR #379 for more details.

3.15.3 2.0.2 - 2021-04-10

Exposes `execute_op()` at top-level namespace

3.15.4 2.0.1 - 2021-04-03

Bug Fixes

- *matmul()* and *multi_matmul()* were missing from the top-level namespace of `mygrad`.
- A 0D tensor involved in a broadcasted operation would have a numpy-float set for its gradient instead of a 0D array.

New Functions

The following non-differentiable NumPy functions now work on mygrad tensors (and return ndarrays). Aliases of these are available at the top-level namespace of `mygrad`

- np.isnan
- np.isfinite
- np.isinf
- np.isnat
- np.signbit
- np.logical_not
- np.logical_and
- np.logical_or
- np.logical_xor

- np.greater
- np.greater_equal
- np.less
- np.less_equal
- np.equal
- np.not_equal
- np.floor_divide
- np.remainder
- np.mod
- np.fmod
- np.divmod
- np.rint
- np.sign
- np.floor
- np.ceil
- np.trunc
- np.isclose

3.15.5 2.0.0 - 2021-03-30

This is a compatibility-breaking update to MyGrad, and it's great! MyGrad 2.0 represents a major overhaul to this project. This release creates near parity between the experiences of using MyGrad and using NumPy, and uses NumPy's new mechanisms for overriding functions so that NumPy functions can operate "directly" on MyGrad's tensors, and thus can be used to construct differentiable computational graphs!

```
>>> import numpy as np
>>> from mygrad import tensor
>>> x = tensor([1., 2.])
>>> np.square(x).backward()  # backprop through NumPy functions!
>>> x.grad
array([2., 4.])
```

Another important, but less exciting, feature is that MyGrad now protects users from inadvertently corrupting the state of a computational graph by, say, mutating a NumPy array that is participating in the graph. This is very useful for protecting people – especially students – from unwittingly poisoning the results of their calculations.

Lastly… no more "nulling" gradients! MyGrad will now handle deleting gradients for you in a way that is nicely compatible with gradient-based optimization work flows.

New Functions and Utilities

- *tensor()*
- *astensor()*
- *asarray()*
- *no_autodiff()*
- *mem_guard_off()*
- *mem_guard_on()*
- *turn_memory_guarding_off()*
- `turn_memory_guarding_on()`
- *concatenate()*
- *stack()*
- *norm()*

Dropping Support for Python 3.6 and Numpy < 1.17

MyGrad now abides by the NEP 29 recommendation, and adopts a common "time window-based" policy for support of Python and NumPy versions.

As such the Python 3.7 and Numpy 1.17 are the minimum versions supported by MyGrad 2.0.

The Interfaces Between `mygrad.Tensor` and `numpy.array` Match

You can now control the dimensionality of a tensor and whether or not a tensor copies its data upon initialization, via the *tensor()* interface. This mirrors the behavior of `array()`

Numpy	MyGrad 1.X	MyGrad 2.0

`>>> np.array([1., 2.],␣`
`↪copy=True, ndmin=2)`
`array([[1., 2.]])`
	`>>> mg.Tensor([1., 2.],␣`
	`↪copy=True, ndmin=2)`
	`<TypeError>`
		`>>> mg.tensor([1., 2.],␣`
		`↪copy=True, ndmin=2)`
		`Tensor([[1., 2.]])`

Support for dtype, where, and out in ufuncs

MyGrad now implements ufuncs with support for specifying dtype, boolean masks, and in-place targets. The additional methods, such as `mygrad.add.reduce`, are not yet implemented.

MyGrad 2.0

`>>> mg.add([1, 2],[0, 2], where=[True, False], dtype=float)`
`Tensor([3., 1.])`

Augmented Updates on Tensors Now Match NumPy's Behavior

Previously, augmented assignment expressions, such as `tensor *= 2`, behaved merely as a shorthand for the simple assignment `tensor = tensor * 2`. This is in stark contrast to the behavior of an augmented assignment on a NumPy array, which mutates the array in-place.

This meant that there was a major discrepancy between how these expressions behaved across MyGrad and NumPy. This has changed in MyGrad 2.0: all augmented assignment expressions operate in-place on tensors and mutate their underlying data.

Numpy	MyGrad 1.X	MyGrad 2.0

`>>> x = np.array([1., 2.])`
`>>> y = x`
`>>> x *= 2`
`>>> x is y`
`True`
	`>>> x = mg.Tensor([1., 2.])`
	`>>> y = x`
	`>>> x *= 2  # x = 2 * x`
	`>>> x is y  # doesn't match!`
	`False`
		`>>> x = mg.tensor([1., 2.])`
		`>>> y = x`
		`>>> x *= 2`
		`>>> x is y  # matches!`
		`True`

Creating and Augmenting Views of Tensors

MyGrad now provides rich support for creating and manipulating views of tensors.

All basic indexing operations performed on a tensor will produce a view of said tensor. This means that these two tensors share memory (While MyGrad 1.X created a view of the underlying NumPy array under the hood for basic indexing, its notion of supporting views went no further than that.) As with NumPy arrays the "parent" of a view can be accessed through the tensor's `.base` attribute

Numpy	MyGrad 1.X	MyGrad 2.0

`>>> x = np.array([1., 2.,␣`
`↪3.])`
`>>> y = x[:2]`
`>>> np.shares_memory(x, y)`
`True`
`>>> y.base is x`
`True`
	`>>> x = mg.Tensor([1., 2.,␣`
	`↪3.])`
	`>>> y = x[:2]`
	`>>> np.shares_memory(x, y)`
	`True`
	`>>> y.base is x  # doesn't␣`
	`↪match!`
	`<AttributeError>`
		`>>> x = mg.tensor([1., 2.,␣`
		`↪3.])`
		`>>> y = x[:2]`
		`>>> np.shares_memory(x, y)`
		`True`
		`>>> y.base is x  # matches!`
		`True`

Mutating shared data will propagate through views:

Numpy	MyGrad 1.X	MyGrad 2.0

`>>> y *= -1`
`>>> y`
`array([-1., -2.])`
`>>> x`
`array([-1., -2., 3.])`
	`>>> y *= -1`
	`>>> y`
	`Tensor([-1., -2.])`
	`>>> x  # doesn't match!`
	`Tensor([1., 2., 3.])`
		`>>> y *= -1`
		`>>> y`
		`Tensor([-1., -2.])`
		`>>> x  # matches!`
		`Tensor([-1., -2., 3.])`

Furthermore, views of tensors now propagate corresponding gradient information as well! This means that if `y` is a view of `x`, then `y.grad` will be a corresponding view of `x.grad`. This is true for all varieties of views, views of views, etc., of x.

```
# Because `y` is a view of `x`, `y.grad` will be
# a corresponding view of `x.grad`
>>> (x ** 2).backward()
>>> x.grad
array([-2., -4.,  6.,  8.])
>>> y.grad
array([-2., -4.])
>>> y.grad.base is x.grad
True
```

This rich support for views, augmented assignments, and in-place updates on tensors enables much more sophisticated operations on tensors now. For example, let's make a shape-(3, 3) tensor and perform and operations involving views of its diagonal and its anti-diagonal. (Note that `einsum()` is capable of returning a view of a tensor's diagonal, and that MyGrad fully supports backpropagation through all flavors of einsum!)

```
>>> x = mg.tensor([[0., 1., 2.],
...                [3., 4., 5.],
...                [6., 7., 8.]])

# view of diagonal of `x`
>>> diag = mg.einsum("ii->i", x)
>>> diag
Tensor([0., 4., 8.])

# view of anti-diagonal of `x`
>>> anti_diag = mg.einsum("ii->i", x[:, ::-1])
>>> anti_diag
Tensor([2., 4., 6.])

# Compute derivatives of their summed difference
>>> (diag - anti_diag).sum().backward()
>>> x.grad
array([[ 1.,  0., -1.],
       [ 0.,  0.,  0.],
       [-1.,  0.,  1.]])

# The views of `x` have the appropriate corresponding
# views of `x.grad`
>>> diag.grad
array([1., 0., 1.])
>>> anti_diag.grad
array([-1.,  0., -1.])
```

Bye-Bye Null Gradients!

Gone are the days of having to manually clear your tensors' gradients and the computational graph that they were in; now MyGrad does it for you! This means that `Tensor.null_gradients()` no longer does anything other than emit a deprecation warning. In an upcoming minor release this method will be removed entirely.

In MyGrad 2.0, calling *`backward()`* will finish its computation by clearing the computational graph that was involved in the backpropagation. Thus any internally-referenced tensors associated with that computational graph become free for garbage collection. This is very nice behavior to help prevent students from filling up their RAM unwittingly.

And instead of worrying about nulling gradients manually, a tensor will automatically have its gradient cleared any time that it is involved in a new mathematical operation. This enables the following common workflow for performing gradient-based optimization:

MyGrad 1.X	MyGrad 2.0
`>>> x = mg.Tensor([1., 2.])`	`>>> x = mg.tensor([1., 2.])`
`>>> for _ in range(10):`	`>>> for _ in range(10):`
`...     y = 3 * x`	`...     y = 3 * x  # nulls grad`
`...     assert x.grad is None`	`...     assert x.grad is None`
`...     y.backward()`	`...     y.backward()`
`...     assert all(x.grad == 3.)`	`...     assert all(x.grad == 3.)`
`...     y.null_gradients()`	

```
for _ in range(num_optimization_steps):
    # using `model_params` in a function will automatically
    # set its gradients to `None`
    loss = compute_loss(data, model_params)  # gradients cleared
    loss.backward()         # compute gradients
    optimize(model_params)  # do stuff with gradients
```

You can also call *`null_grad()`* to manually clear an individual tensor's gradient.

Safety First: Memory Guarding Behavior in MyGrad 2.0

In MyGrad 1.X it was all too easy to unwittingly corrupt the state of a computational graph by mutating a NumPy array mid-computation. This could lead to incorrect calculations of gradients! This is the stuff of horrifying nightmares.

Now MyGrad tracks all of the arrays that are involved in active computational graphs and locks their memory so that they are read-only (except for when the user mutates the array explicitly with a MyGrad operation). This means that the sort of mutation that could have lurked silently in the dimly-lit alleyways of bugs-ville will now get loudly narc'd on by MyGrad's merciless memory guard!

MyGrad 1.X	MyGrad 2.0
`>>> arr = np.array([1., 2.])`	`>>> arr = np.array([1., 2.])`
`>>> tn = mg.Tensor([1. 1.])`	`>>> tn = mg.tensor([1. 1.])`
`>>> z = x * y`	`>>> z = x * y`
`# mutating x will corrupt`	`# mutating x will corrupt`
`# backprop through z...`	`# backprop through z...`
`>>> x[:] = 0.`	`>>> x[:] = 0. # you shall not pass!`
	`ValueError: read-only!`
`>>> z.backward() # uh oh...`	`>>> z.backward()`
`>>> tn.grad # should be: (1., 2.)`	`>>> tn.grad`
`array([0., 0.])`	`array([1., 2.])`

Any tensor or array that is no longer participating in an active computational graph will automatically have its write-ability restored to its original state.

```
# memory guarding is released once an array is no
# longer involved in an active computational graph
>>> import mygrad as mg
>>> import numpy as np
>>> x = np.array([1., 2.])
>>> y = mg.ones_like(x)
>>> z = x * y     # x and y are locked
>>> z.backward()  # graph cleared; x and y are "released"
>>> x[:] = 0      # can write to x
>>> x
array([0., 0.])

# This result is not referenced, thus
# x and y are immediately released by the
# memory-guard; no graph-clearing is needed
>>> x * y
Tensor([0., 0.])
>>> x[:] = 1.
```

But with great responsibility comes great . . . uhh. . . slowness? This memory-guarding feature can lead to slowdowns of **up to 50% for computations involving many small tensors** (It used to be **a lot** worse. . . like 5x worse. I worked really hard to speed it up! I promise!). That being said, computations involving beefy tensors (e.g. standard neural networks) will not be significantly affected by the overhead associated with the memory guard. Please refer to *Performance Tips* for responsible ways to disable this memory-guarding mechanism.

Speaking of optimizations. . .

Disabling Automatic Differentiation

Sometimes you want to use your MyGrad code to do calculations, but you don't actually need to compute any derivatives. A common example of this is evaluating the test-time performance of a machine learning model that you are in the process of optimizing – you don't actually need to perform backpropagation when you are processing the test data.

In these circumstances, you can greatly reduce the overhead cost associated with building a computational graph by using the `no_autodiff()` decorator / context manager. See the linked documentation for extensive examples of its usage.

```
# demonstrating mygrad in no-autodiff mode
>>> import mygrad as mg
>>> x = mg.Tensor([1., 2., 3., 4.])
>>> with mg.no_autodiff:
...     y = x ** 2  # operation not tracked
>>> y.backward()
>>> y.grad, x.grad  # x is not "connected" to y
(array([1., 1., 1.]), None)
```

For computations involving many small tensors, this can produce **up to a 3x speedup**! So make sure you make keen use of this when you don't actually need to perform autodiff.

Revamping Constant Semantics to be Explicit

Previously, specifying `constant=False` in a mygrad function did not actually mean that the function would necessarily produce a non-constant tensor. Rather, it simply meant that the output would not be _forced_ to be a constant – whether or not the result was a constant depended on the inputs (i.e. a function whose inputs were all constants would thus produce a constant).

This was a very bad design decision! Now, specifying `constant=False` guarantees that the output of a function is a non-constant (meaning that it facilitates backpropagation through a computational graph).

That being said, we usually _do_ want constant information to propagate through functions. Thus `constant=None` is now the default value – its behavior matches that of `constant=False` from MyGrad 1.X – for all functions that accept the argument.

It is also now standard to require that this argument be a keyword-only argument.

MyGrad 1.X	MyGrad 2.0

```
>>> t1 = mg.tensor(1., constant=True)
>>> t2 = mg.tensor(1., constant=True)

>>> out = mg.add(t1, t2, constant=False)
>>> out.constant
True
```
	```
	>>> t1 = mg.tensor(1., constant=True)
	>>> t2 = mg.tensor(1., constant=True)

	>>> out = mg.add(t1, t2, constant=False)
	>>> out.constant
	False

	# constant = None
	>>> out = mg.add(t1, t2)
	>>> out.constant
	True
	```

```
>>> t1 = mg.tensor(1., constant=True)
>>> t2 = mg.tensor(1., constant=True)
```

# old behavior >>> out = mg.add(t1, t2, constant=False) >>> out.constant True

# new behavior >>> out = mg.add(t1, t2, constant=False) >>> out.constant False

```
>>> out = mg.add(t1, t2, constant=None)
>>> out.constant
True
```

Remove Scalar-Only Conditions on Backpropagation

Previously, one could only invoke backpropagation from a non-scalar tensor only if that tensor was the culmination of operations that preserved a one-to-one mapping between the elements of an upstream tensor with its downstream neighbor. Otherwise an error was raised. This ensured that `tensor.grad` would always be the same shape as `tensor`, and not represent a higher-dimensional tensor.

Now calling `tensor.backward()` from a non-scalar tensor will behave as if the tensor was summed prior to invoking backpropagation. This is simple, easy-to-understand behavior, which ensures that `tensor.grad` can always be interpreted as an array of scalar-valued derivatives.

MyGrad 1.X	MyGrad 2.0

`>>> t1 = mg.Tensor([[1., 2.],`
`...                  [0., -1]])`
`>>> t2 = mg.Tensor([[0., 1.],`
`...                  [3., -1]])`
`>>> z = t1 @ t2`
`>>> z.backward()`
`<InvalidBackprop: Scalar-only>`
	`>>> t1 = mg.tensor([[1., 2.],`
	`...                  [0., -1]])`
	`>>> t2 = mg.tensor([[0., 1.],`
	`...                  [3., -1]])`
	`>>> z = t1 @ t2`
	`>>> z.backward()`
	`>>> t1.grad`
	`array([[1., 2.],`
	`       [1., 2.]])`

Integer-valued Tensors Are Treated as Constants

Derivatives involving integer-valued tensors are typically ill-defined, and in MyGrad 1.X they were generally just wrong. Now integer-valued tensors can only be involved in computational graphs as constants.

MyGrad 1.X	MyGrad 2.0

`>>> t1 = mg.Tensor([[1, 2]).constant`
`False`
	`>>> t1 = mg.tensor([[1, 2]]).constant`
	`True`

Is This Code Well-Tested?

Yes! I consider MyGrad's test suite to be the most important part of the library. It is the only reason why I feel comfortable releasing this code for students, teachers, and others to use. I leverage thorough property-based testing using the Hypothesis library to exercise this code as rigorously as I can manage. These tests even found bugs in NumPy!

Special Thanks

Special thanks to Alex Silverstein, Zac Dodds, and Petar Griggs for all of the fruitful discussions, ideas, and influence that you provided throughout this major update.

3.15.6 1.9.0 - 2020-08-28

The most significant aspect of this release is the implementation of `Tensor.__array__`, which enables a huge amount of cross-compatibility with numpy utilities (#288). Note that any previous reliance of a numpy function to produce an array of tensor-scalars will likely produce a standard numpy array instead.

Improvements:

- `x**1` and `x**2` are now special-cased in order to make these common operations more efficient (#266)
- The derivative of *`focal_loss()`* was refactored to handle special edge-cases and the tests for focal loss were improved to exercise these edge cases (#269)
- Various improvements to the tests (#271, #277, #290, #284, #289, #282, #292, #293)
- The internal mechanism for tracking tensors in computational graph now depends on hashing tensor-IDs instead of hashing tensors directly. The fact that tensors could be hashed was due to the fact that its equality specialty methods were being monkey-patched (#276)
- *`softmax()`* and *`logsoftmax()`* both expose `axis` arguments (#268)

Bug fixes:

- 0D tensors could not be indexed into – e.g. to insert a newaxis (#273)
- There was a potential numerical instability in *`mygrad.nnet.layers.batchnorm()`* (#285)
- The `dtype` argument in `Tensor.__init__` was ignored when the array-like argument, x, was another Tensor-instance (#294)

New features:

- `Tensor.__array__` now exposes the tensor's underlying numpy array – this enables a huge amount of cross-compatibility with numpy utilities (#288)
- Adds *`asarray()`* (#279)
- Adds *`astensor()`* (#294)

3.15.7 1.8.1 - 2020-07-28

This is an internal change to the backprop mechanism for `Tensor.__getitem__`, which produces considerable speedups (2x-4x) for backprop through basic indexing and boolean indexing. Thanks to Petar Griggs for finding this.

3.15.8 1.8.0 - 2020-07-25

New features:

- Adds `any()` and `any()`
- Adds `rand()`
- Adds `randint()`
- Adds `randn()`
- Adds `random()`
- Adds `random_integers()`
- Adds `random_sample()`
- Adds `ranf()`
- Adds `sample()`
- Adds `seed()`

Thanks to Darshan Krishnaswamy and Sam Carpenter for adding this functionality!

Fixes a bug in the GRU layer where mixed floating point precision dtypes between data and weights raised an error. Thanks to Petar Griggs for the fix!

3.15.9 1.7.1 - 2020-07-11

Fixes a bug in *`negative_log_likelihood()`*, where setting `constant=True` had no effect.

3.15.10 1.7.0 - 2020-07-11

This release continues the process of integrating functions from mynn.

New features:

- Adds *`glorot_normal()`*
- Adds *`glorot_uniform()`*
- Adds *`he_normal()`*
- Adds *`he_uniform()`*
- Adds *`normal()`*
- Adds *`uniform()`*
- Adds *`focal_loss()`*
- Adds *`negative_log_likelihood()`*

Big thanks to David Mascharka!

Improvements:

The interfaces to *reshape()* and reshape() were adjusted to match exactly the interfaces to their NumPy counterparts. I.e. *reshape()* now requires newshape to be a sequence, whereas reshape() can accept an unpacked sequence for its newshape.

shape() is now settable - triggering an in-place reshape of a tensor, matching the corresponding behavior in NumPy.

Internal changes:

The logic for writing an in-place operation has been consolidated into a convenient wrapper: _in_place_op().

3.15.11 1.6.0 - 2020-06-21

New features:

- Adds *elu()*
- Adds *glu()*
- Adds *leaky_relu()*
- Adds *selu()*
- Adds *soft_sign()*

Big thanks to David Mascharka!

3.15.12 1.5.0 - 2020-02-16

New features:

- Adds *astype()* method.
- Adds *hard_tanh()*
- y_true can now be passed as a Tensor to *softmax_crossentropy()*

This update also includes various improvements to the library's test suite.

3.15.13 1.4.1 - 2020-01-09

This release performs an internal refactor in the nnet module of the library, as well as an analogous refactor in the test suite. This also fixes a docstring in the multiclass_hinge loss to properly show a description in the readthedocs page.

3.15.14 1.4.0 - 2019-12-19

This release adds the *repeat()* operation. It also includes some minor improvements to mygrad's test suite.

3.15.15 1.3.0 - 2019-11-30

This release adds `clip()` and `where()`.

It also includes a major fix to the graph-traversal mechanism for null-gradients and clear-graph, eliminating an exponentially-scaling runtime.

`+x` will now invoke `mygrad.positive`, mirroring the numpy behavior

There are improvements to user-facing error messages and input validation in addition to major improvements to mygrad's test suite. There is now a 100% line-coverage gate in mygrad's CI system.

3.15.16 1.2.0 - 2019-08-03

We're finally keeping a formal changelog!

This release makes substantial improvements to MyGrad's error-checking and handling, in order to make much simpler the process of debugging issues with buggy custom operations. Specifically, `backward()` now checks for an invalid-gradients on each call of `backward_var()`, and raises a descriptive error message.

`mygrad.errors` was introduced to provide descriptive, MyGrad-specific exceptions. For example, we no longer raise bare exceptions for scenarios like invalid backprop through a scalar-only graph; rather, we now raise a descriptive `InvalidBackprop` exception.

MyGrad's testing framework received wide-ranging improvements, yielding complete test coverage and fewer flaky tests. Coverage checks were added to the project's CI process.

`maximum()` and `minimum()` were patched to permit backpropagation through scalar inputs.

Internal implementation details of `einsum()` were adjusted to remove redundant code in its backpropagation machinery.

`null_gradients()` was refactored to ensure that only a single traversal of the computational graph is performed to null all of the tensors' gradients. Furthermore, *Tensor.null_gradients(clear_graph=True)* now only performs a single graph traversal, instead of two.

In keeping with NumPy's behavior, performing $+x$ (where x is a mygrad-tensor) no longer returns a reference of x, but returns *mygrad.positive(x)*.

Backpropagation through `max()` and `min()` now works for 0D tensors.

Input validation was added to `mygrad.nnet.layers.utils.sliding_window_view()`.

Fixed backpropagation through basic indexing, $x[ind] = b$, in which broadcasting occurred and b possess "excess" leading singleton dimensions.

BIBLIOGRAPHY

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.ones.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.ones_like.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.zeros.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.zeros_like.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.eye.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.identity.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.full.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.full_like.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.empty.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.empty_like.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arange.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.logspace.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.geomspace.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.stack.html

[1] Cormen, “Introduction to Algorithms”, Chapter 15.2, p. 370-378

[2] http://en.wikipedia.org/wiki/Matrix_chain_multiplication

[1] Retrieved from: https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.sin.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.cos.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.tan.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arcsin.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arccos.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arctan.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arctan.html

[2] ISO/IEC standard 9899:1999, “Programming language C.”

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.sinh.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.cosh.html

[1] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Hyperbolic function”, https://en.wikipedia.org/wiki/Hyperbolic_function

[3] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.tanh.html

[1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Inverse hyperbolic function”, https://en.wikipedia.org/wiki/Arcsinh

[3] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arcsinh.html

[1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Inverse hyperbolic function”, https://en.wikipedia.org/wiki/Arccosh

[3] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arccosh.html

[1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Inverse hyperbolic function”, https://en.wikipedia.org/wiki/Arctanh

[3] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.arctanh.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.exp.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.expm1.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.exp2.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.log.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.log10.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.log2.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.log1p.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.logaddexp.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.logaddexp2.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.add.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.reciprocal.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.positive.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.negative.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.multiply.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.power.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.subtract.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.sqrt.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.cbrt.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.square.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.absolute.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.maximum.html

[1] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.minimum.html

[1] S. Merity, et. al. "Regularizing and Optimizing LSTM Language Models", arXiv:1708.02182v1, 2017.

[2] Y. Gal, Z. Ghahramani "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks" arXiv:1512.05287v5, 2016.

[1] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, Sepp Hochreiter Self-Normalizing Neural Networks https://arxiv.org/abs/1706.02515

[1] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Hyperbolic function", https://en.wikipedia.org/wiki/Hyperbolic_function

[3] Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.tanh.html

INDEX

Symbols

`__init__()` (*mygrad.absolute method*), 176
`__init__()` (*mygrad.add method*), 153
`__init__()` (*mygrad.arccos method*), 92
`__init__()` (*mygrad.arccosh method*), 111
`__init__()` (*mygrad.arcsin method*), 90
`__init__()` (*mygrad.arcsinh method*), 108
`__init__()` (*mygrad.arctan method*), 95
`__init__()` (*mygrad.arctan2 method*), 97
`__init__()` (*mygrad.arctanh method*), 114
`__init__()` (*mygrad.cbrt method*), 170
`__init__()` (*mygrad.cos method*), 86
`__init__()` (*mygrad.cosh method*), 102
`__init__()` (*mygrad.exp method*), 131
`__init__()` (*mygrad.exp2 method*), 135
`__init__()` (*mygrad.expm1 method*), 133
`__init__()` (*mygrad.log method*), 137
`__init__()` (*mygrad.log10 method*), 139
`__init__()` (*mygrad.log1p method*), 143
`__init__()` (*mygrad.log2 method*), 141
`__init__()` (*mygrad.logaddexp method*), 146
`__init__()` (*mygrad.logaddexp2 method*), 148
`__init__()` (*mygrad.matmul method*), 73
`__init__()` (*mygrad.maximum method*), 178
`__init__()` (*mygrad.minimum method*), 180
`__init__()` (*mygrad.multiply method*), 160
`__init__()` (*mygrad.negative method*), 158
`__init__()` (*mygrad.nnet.activations.tanh method*), 214
`__init__()` (*mygrad.operation_base.Operation method*), 33
`__init__()` (*mygrad.positive method*), 156
`__init__()` (*mygrad.power method*), 163
`__init__()` (*mygrad.reciprocal method*), 155
`__init__()` (*mygrad.sin method*), 84
`__init__()` (*mygrad.sinh method*), 99
`__init__()` (*mygrad.sqrt method*), 169
`__init__()` (*mygrad.square method*), 172
`__init__()` (*mygrad.subtract method*), 165
`__init__()` (*mygrad.tan method*), 88
`__init__()` (*mygrad.tanh method*), 105

A

`absolute` (*class in mygrad*), 173
`add` (*class in mygrad*), 151
`add_sequence()` (*in module mygrad*), 149
`amax()` (*in module mygrad*), 125
`amin()` (*in module mygrad*), 126
`arange()` (*in module mygrad*), 50
`arccos` (*class in mygrad*), 91
`arccosh` (*class in mygrad*), 110
`arcsin` (*class in mygrad*), 88
`arcsinh` (*class in mygrad*), 107
`arctan` (*class in mygrad*), 93
`arctan2` (*class in mygrad*), 95
`arctanh` (*class in mygrad*), 113
`asarray()` (*in module mygrad*), 36
`astensor()` (*in module mygrad*), 37
`astype()` (*mygrad.Tensor method*), 14
`atleast_1d()` (*in module mygrad*), 61
`atleast_2d()` (*in module mygrad*), 62
`atleast_3d()` (*in module mygrad*), 63

B

`backward()` (*mygrad.operation_base.Operation method*), 33
`backward()` (*mygrad.Tensor method*), 15
`backward_var()` (*mygrad.operation_base.Operation method*), 33
`base` (*mygrad.Tensor property*), 17
`batchnorm()` (*in module mygrad.nnet.layers*), 183
`broadcast_to()` (*in module mygrad*), 65
`build_graph()` (*in module mygrad.computational_graph*), 226

C

`cbrt` (*class in mygrad*), 169
`clear_graph()` (*mygrad.Tensor method*), 17
`clip()` (*in module mygrad*), 166
`concatenate()` (*in module mygrad*), 67
`constant` (*mygrad.Tensor property*), 18
`conv_nd()` (*in module mygrad.nnet.layers*), 184
`copy()` (*mygrad.Tensor method*), 19
`cos` (*class in mygrad*), 84

cosh (*class in mygrad*), 101
creator (*mygrad.Tensor property*), 19
cumprod() (*in module mygrad*), 119
cumsum() (*in module mygrad*), 120

D

dirac() (*in module mygrad.nnet.initializers*), 215
divide (*in module mygrad*), 161
dtype (*mygrad.Tensor property*), 20

E

einsum() (*in module mygrad*), 75
elu() (*in module mygrad.nnet.activations*), 199
empty() (*in module mygrad*), 47
empty_like() (*in module mygrad*), 48
exp (*class in mygrad*), 129
exp2 (*class in mygrad*), 134
expand_dims() (*in module mygrad*), 65
expm1 (*class in mygrad*), 132
eye() (*in module mygrad*), 44

F

flatten() (*mygrad.Tensor method*), 57
focal_loss() (*in module mygrad.nnet.losses*), 193
full() (*in module mygrad*), 45
full_like() (*in module mygrad*), 46

G

geomspace() (*in module mygrad*), 53
glorot_normal() (*in module mygrad.nnet.initializers*), 216
glorot_uniform() (*in module mygrad.nnet.initializers*), 216
glu() (*in module mygrad.nnet.activations*), 200
grad (*mygrad.Tensor property*), 20
gru() (*in module mygrad.nnet.layers*), 191

H

hard_tanh() (*in module mygrad.nnet.activations*), 201
he_normal() (*in module mygrad.nnet.initializers*), 217
he_uniform() (*in module mygrad.nnet.initializers*), 218

I

identity() (*in module mygrad*), 45
item() (*mygrad.Tensor method*), 21

L

leaky_relu() (*in module mygrad.nnet.activations*), 203
linspace() (*in module mygrad*), 51
load() (*in module mygrad*), 225
log (*class in mygrad*), 136
log10 (*class in mygrad*), 138
log1p (*class in mygrad*), 142
log2 (*class in mygrad*), 140
logaddexp (*class in mygrad*), 144
logaddexp2 (*class in mygrad*), 146
logsoftmax() (*in module mygrad.nnet.activations*), 203
logspace() (*in module mygrad*), 52

M

margin_ranking_loss() (*in module mygrad.nnet.losses*), 194
matmul (*class in mygrad*), 71
max() (*in module mygrad*), 127
max_pool() (*in module mygrad.nnet.layers*), 189
maximum (*class in mygrad*), 176
mean() (*in module mygrad*), 121
mem_guard_off (*in module mygrad*), 27
mem_guard_on (*in module mygrad*), 28
min() (*in module mygrad*), 128
minimum (*class in mygrad*), 179
moveaxis() (*in module mygrad*), 58
multi_matmul() (*in module mygrad*), 74
multiclass_hinge() (*in module mygrad.nnet.losses*), 194
multiply (*class in mygrad*), 159
multiply_sequence() (*in module mygrad*), 150

N

ndim (*mygrad.Tensor property*), 22
negative (*class in mygrad*), 157
negative_log_likelihood() (*in module mygrad.nnet.losses*), 195
no_autodiff (*in module mygrad*), 26
norm() (*in module mygrad.linalg*), 80
normal() (*in module mygrad.nnet.initializers*), 219
null_grad() (*mygrad.Tensor method*), 22
null_gradients() (*mygrad.Tensor method*), 23

O

ones() (*in module mygrad*), 39
ones_like() (*in module mygrad*), 40
Operation (*class in mygrad.operation_base*), 32

P

positive (*class in mygrad*), 155
power (*class in mygrad*), 161
prod() (*in module mygrad*), 117

R

ravel() (*in module mygrad*), 55
reciprocal (*class in mygrad*), 153
relu() (*in module mygrad.nnet.activations*), 210
repeat() (*in module mygrad*), 70
reshape() (*in module mygrad*), 56
roll() (*in module mygrad*), 58

S

save() (*in module mygrad*), 224
selu() (*in module mygrad.nnet.activations*), 205
shape (*mygrad.Tensor property*), 23
sigmoid() (*in module mygrad.nnet.activations*), 207
sin (*class in mygrad*), 82
sinc() (*in module mygrad*), 151
sinh (*class in mygrad*), 98
size (*mygrad.Tensor property*), 24
sliding_window_view() (*in module mygrad*), 221
soft_sign() (*in module mygrad.nnet.activations*), 209
softmax() (*in module mygrad.nnet.activations*), 207
softmax_crossentropy() (*in module mygrad.nnet.losses*), 196
softmax_focal_loss() (*in module mygrad.nnet.losses*), 198
sqrt (*class in mygrad*), 167
square (*class in mygrad*), 171
squeeze() (*in module mygrad*), 66
stack() (*in module mygrad*), 68
std() (*in module mygrad*), 124
subtract (*class in mygrad*), 164
sum() (*in module mygrad*), 118
swapaxes() (*in module mygrad*), 59

T

T (*mygrad.Tensor property*), 24
tan (*class in mygrad*), 86
tanh (*class in mygrad*), 104
tanh (*class in mygrad.nnet.activations*), 212
tensor() (*in module mygrad*), 34
transpose() (*in module mygrad*), 60
turn_memory_guarding_off() (*in module mygrad*), 28

U

uniform() (*in module mygrad.nnet.initializers*), 220

V

var() (*in module mygrad*), 122

W

where() (*in module mygrad*), 181

Z

zeros() (*in module mygrad*), 41
zeros_like() (*in module mygrad*), 42