TensorNN

Release 0.0.1

Arjun Sahlot

Apr 20, 2023

CONTENTS

A python machine learning library made from scratch.

CHAPTER

ONE

TENSORNN

tensornn.activation	This file contains the activation functions of TensorNN.
tensornn.errors	This file contains errors the TensorNN might raise.
tensornn.layers	This file contains different types of layers used in neural networks.
tensornn.loss	This file contains the loss functions used in TensorNN.
tensornn.nn	This file contains the neural network class.
tensornn.optimizers	This file contains optimizers that help tune your neural network.
tensornn.tensor	This file contains the Tensor class for TensorNN.
tensornn.utils	This file contains useful variables that are used in TensorNN.

1.1 tensornn.activation

This file contains the activation functions of TensorNN. Activation functions modify their input to create non-linearity in the network. This allows your network to handle more complex problems. They are very similar to a layer.

Classes

Activation	Base activation class.
ELU	Exponential linear unit is similar to ReLU, but it is not piecewise.
LeakyReLU	Leaky ReLU is extremely similar to ReLU.
NewtonsSerpentine	Haven't seen it anywhere so I am not sure if this is good but seemed like a good candidate.
NoActivation	Linear activation function, doesn't change anything.
ReLU	The rectified linear unit activation function is one of the simplest activation function.
Sigmoid	The sigmoid function's output is always between -1 and 1 Formula: `1 / (1+e^(-x))` | constants: e(Euler's number, 2.718...)
Softmax	The softmax activation function is most commonly used in the output layer.
Swish	The swish activation function is the output of the sigmoid function multiplied by x.
Tanh	TODO: add description

1.1.1 tensornn.activation.Activation

class tensornn.activation.**Activation**

Bases: ABC

Base activation class. All activation classes should inherit from this.

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

abstract derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

abstract forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.2 tensornn.activation.ELU

class tensornn.activation.**ELU**(*a: float = 1*)

Bases: *Activation*

Exponential linear unit is similar to ReLU, but it is not piecewise. Formula: `A*((e^x)-1)` | constants: A, e(Euler's number, 2.718...)

Ex, A=1: `[12.319, -91.3, 0.132] -> [12.319, -1, 0.132]`

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__init__(*a: float = 1*) → None

Initialize ELU.

Parameters
a – multiplier used in formula, checkout help(tnn.activation.ELU), defaults to 1

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.3 tensornn.activation.LeakyReLU

class tensornn.activation.**LeakyReLU**(*a: float = 0.1*)

Bases: *Activation*

Leaky ReLU is extremely similar to ReLU. ReLU is LeakyReLU if A was 1. Formula: `if x>=0, x; if x<0, Ax` | constants: A(leak)

Ex, A=0.1: `[12.319, -91.3, 0.132] -> [12.319, -9.13, 0.132]`

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__init__(*a: float = 0.1*) → None

Initialize LeakyReLU.

Parameters
a – multiplier used in formula, checkout help(tnn.activation.LeakyReLU), defaults to 1

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.4 tensornn.activation.NewtonsSerpentine

class tensornn.activation.**NewtonsSerpentine**(*a: float = 1*, *b: float = 1*)

Bases: *Activation*

Haven't seen it anywhere so I am not sure if this is good but seemed like a good candidate. NOTE: THIS IS NOT A GOOD CANDIDATE. Larger numbers result in a lower value, which means being large doesn't give importance. Do not use unless you want to have some fun ;)

Formula: `(A*B*x)/(x^2+A^2)` | A, B constants

Ex, A=1,B=1: `[12.319, -91.3, 0.132] -> [0.08064402, -0.01095159, 0.12973942]`

https://mathworld.wolfram.com/SerpentineCurve.html

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__init__(*a: float = 1*, *b: float = 1*) → None

Initialize Newton's Serpentine. Checkout the formula by using: help(tensornn.activation.NewtonsSerpentine)

Parameters

- **a** – constant in equation, defaults to 1
- **b** – constant in equation, defaults to 1

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.5 tensornn.activation.NoActivation

class tensornn.activation.**NoActivation**

Bases: *Activation*

Linear activation function, doesn't change anything. Use this if you don't want an activation.

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.6 tensornn.activation.ReLU

class tensornn.activation.**ReLU**

Bases: *Activation*

The rectified linear unit activation function is one of the simplest activation function. It is a piecewise function.
Formula: `if x>=0, x; if x<0, 0`

Ex: `[12.319, -91.3, 0.132] -> [12.319, 0, 0.132]`

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.7 tensornn.activation.Sigmoid

class tensornn.activation.**Sigmoid**

Bases: *Activation*

The sigmoid function's output is always between -1 and 1 Formula: `1 / (1+e^(-x))` | constants: e(Euler's number, 2.718...)

Ex: `[12.319, -91.3, 0.132] -> [9.99995534e-01, 2.23312895e-40, 5.32952167e-01]`

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.8 tensornn.activation.Softmax

class tensornn.activation.**Softmax**

Bases: *Activation*

The softmax activation function is most commonly used in the output layer. If you are using this activation function, you should be using tnn.CategoricalCrossEntropy as your loss function. This is because the softmax function always generates a probability distribution with all values between 0 and 1, and for these types of values, tnn.CategoricalCrossEntropy is the best loss function to use.

The goal of softmax is to convert the predicted values of the network into percentages that add up to 1. Ex. it converts [-1.42, 3.312, 0.192] to [0.00835, 0.94970, 0.41935] which is much easier to understand.

When coming up with a way to write this, a big problem is negative numbers since we can't have negative numbers in our final output. So how do we get rid of them? Do we clip them to 0? Do we square them? Do we use absolute value? Though all these methods seem nice, they take away from the value of negative numbers. If we clip to 0 then negative numbers are no more than just 0, and squaring or using absolute value will just result in the opposite of what we want (large negative number turns into large positive number). So the most effective way is to use exponentiation. Through exponentiation, negative numbers will be small while positive numbers will be large.

But exponentiation raises a new problem, super large numbers which can cause overflow. Fortunately there is a simple solution, we can convert all the values into non positive values prior to exponentiation. We can do this by subtracting each value by the maximum value of our output. This way our values before exponentiation will range between -inf to 0 and our values after exponentiation will range between 0 (e^-inf) to 1 (e^0).

Finally, to come up with all the percentages we can just figure out how much each value contributes to the final sum, what fraction of the sum does each value make. So we can do each value divided by the total sum.

All steps/TLDR: Starting values (from previous example): [-1.42, 3.312, 0.192] Subtract largest value to make all negative: 3.312 is max so subtract from all values, [-4.732, 0, -3.120] Exponentiation, raise each value to e (e^x): [0.0080884, 1, 0.04415717] Come up with percentages, divide each number by the sum: sum is 1.05224557 so we divide each value by it, [0.00836574, 0.94969828, 0.04193599]

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.9 tensornn.activation.Swish

class tensornn.activation.**Swish**

Bases: *Activation*

The swish activation function is the output of the sigmoid function multiplied by x. Formula: `x / (1+e^(-x))` | constants: e(Euler's number, 2.718...)

Ex: `[12.319, -91.3, 0.132] -> [1.23189450e+01, -2.03884673e-38, 7.03496861e-02]`

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.1.10 tensornn.activation.Tanh

class tensornn.activation.**Tanh**

Bases: *Activation*

TODO: add description

Methods

derivative	The derivative of the function.
forward	Calculate a forwards pass of this activation function.

__repr__() → str

Return repr(self).

derivative(*inputs:* Tensor) → *Tensor*

The derivative of the function. Used for backpropagation.

Parameters
inputs – get the derivative of the function at this input

Returns
the derivative of the function at the given input

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this activation function.

Parameters
inputs – the outputs from the previous layer

Returns
the inputs after they are passed through the activation function

1.2 tensornn.errors

This file contains errors the TensorNN might raise.

Exceptions

InitializationError	Raised when there is a problem with initialization, ex: too few layers
InputDimError	Raised when number of dimensions of inputs is not correct.
NotRegisteredError	Raised when you try to train your NeuralNetwork before registering it.

1.2.1 tensornn.errors.InitializationError

exception tensornn.errors.**InitializationError**

Raised when there is a problem with initialization, ex: too few layers

1.2.2 tensornn.errors.InputDimError

exception tensornn.errors.**InputDimError**

Raised when number of dimensions of inputs is not correct.

1.2.3 tensornn.errors.NotRegisteredError

exception tensornn.errors.**NotRegisteredError**

Raised when you try to train your NeuralNetwork before registering it.

1.3 tensornn.layers

This file contains different types of layers used in neural networks. Layers need to be able to propagate their inputs forward.

Functions

flatten	Flatten the inputs array.

1.3.1 tensornn.layers.flatten

tensornn.layers.**flatten**(*inputs:* Tensor) → *Tensor*

Flatten the inputs array. For example, a neural network cannot take in an image as input since it is 2D, so we can flatten it to make it 1D. This should be the first layer of the network.

Classes

Dense	Each neuron is connected to all neurons in the previous layer.
Layer	Abstract base layer class.

1.3.2 tensornn.layers.Dense

class tensornn.layers.**Dense**(*num_neurons: int*, *num_inputs: Optional[int] = None*, *activation:* Activation = *TensorNN.NoActivation*, *zero_biases: bool = True*)

Bases: *Layer*

Each neuron is connected to all neurons in the previous layer. Output is calculated by: (output of previous layer * weights) + biases.

Methods

backward	
forward	Calculate a forwards pass of this layer, before and after activation.
register	Number of inputs in the previous layer.

Attributes

__init__(*num_neurons: int*, *num_inputs: Optional[int] = None*, *activation:* Activation = *TensorNN.NoActivation*, *zero_biases: bool = True*) → None

Initialize dense layer.

Parameters

- **num_neurons** – the number of neurons in this layer/number of outputs of this layer
- **num_inputs** – if this is the first layer, then num_inputs must be filled out
- **activation** – the activation function applied before the layer output is calculated
- **zero_biases** – whether or not the biases should be initialized to 0, if your network dies try setting this to False

#TODO have stuff like zero_biases go in a dictionary like config or options

__repr__() → str

Return repr(self).

forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this layer, before and after activation.

Parameters
inputs – outputs from the previous layer

Returns
the output calculated after this layer before and after activation

register(*prev: int*) → None

Number of inputs in the previous layer. This is called whenever the NeuralNetwork is registered(NeuralNetwork.register()) with the optimizer and loss, it calls this method for all layers giving information to it. If your layer doesn't need this, you don't need to implement this.

Parameters
prev – number of neurons in previous layer

Returns
Nothing

1.3.3 tensornn.layers.Layer

class tensornn.layers.**Layer**(*num_neurons: int*, *num_inputs: Optional[int] = None*, *activation:* Activation = *TensorNN.NoActivation*)

Bases: ABC

Abstract base layer class. All layer classes should inherit from this.

A neural network is composed of layers. A set of inputs are moved from one layer to another. Each layer has its own way of calculating the output of its own inputs(outputs of previous layer). Some layers also have a few tweakable parameters, tweaking these parameters will allow the network to learn and adapt to the inputs to produce the correct outputs.

Methods

forward	Calculate a forwards pass of this layer, before and after activation.
register	Number of inputs in the previous layer.

Attributes

neurons

__init__(*num_neurons: int*, *num_inputs: Optional[int] = None*, *activation:* Activation = *TensorNN.NoActivation*) → None

Initialize a TensorNN Layer

Parameters

- **num_neurons** – the number of neurons in this layer/number of outputs of this layer
- **num_inputs** – if this is the first layer, then num_inputs must be filled out
- **activation** – the activation function applied before the layer output is calculated, defaults to NoActivation

abstract forward(*inputs:* Tensor) → *Tensor*

Calculate a forwards pass of this layer, before and after activation.

Parameters
inputs – outputs from the previous layer

Returns
the output calculated after this layer before and after activation

register(*prev: int*) → None

Number of inputs in the previous layer. This is called whenever the NeuralNetwork is registered(NeuralNetwork.register()) with the optimizer and loss, it calls this method for all layers giving information to it. If your layer doesn't need this, you don't need to implement this.

Parameters
prev – number of neurons in previous layer

Returns
Nothing

1.4 tensornn.loss

This file contains the loss functions used in TensorNN. Loss functions are ways your neural network calculates how off its calculations are. Then this information is used to improve/train it.

Classes

BinaryCrossEntropy	Sigmoid is the only activation function compatible with BinaryCrossEntropy loss.
CategoricalCrossEntropy	It is recommended to use the Softmax activation function with this loss.
Loss	Base loss class.
MAE	Mean absolute error is MSE but instead of squaring the values, you absolute value them.
MSE	Mean squared error is calculated extremely simply.
MSLE	Mean squared logarithmic error is MSE but taking the log of our values before subtraction.
Poisson	Poisson loss is calculated with this formula: average of (pred-desired*log(pred))
RMSE	Root mean squared error is just MSE, but it includes a square root after taking the average.
RSS	Residual sum of squares loss is MSE but instead of doing the mean, you do the sum.
SquaredHinge	Square hinge loss is calculated with this formula: max(0, 1-pred*desired)^2

1.4.1 tensornn.loss.BinaryCrossEntropy

class tensornn.loss.**BinaryCrossEntropy**

Bases: *Loss*

Sigmoid is the only activation function compatible with BinaryCrossEntropy loss. This is how it is calculated: `-(desired*log(pred) + (1-desired)*log(1-pred))`.

Note: log in programming is usually `log` or `natural log` or `ln` in math

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.2 tensornn.loss.CategoricalCrossEntropy

class tensornn.loss.**CategoricalCrossEntropy**

Bases: *Loss*

It is recommended to use the Softmax activation function with this loss. Despite its long name, the way that categorical cross entropy loss is calculated is simple.

Let's say our prediction (after softmax) is `[0.7, 0.2, 0.1]`, and the desired values are `[1, 0, 0]`. We can simply get the prediction number at the index of the `1` in the desired values. 1 is at index 0 so we look at index 0 of our prediction which would be `0.7`. Now we just take the negative log of `0.7` and we are done!

Note: log in programming is usually `log` or `natural log` or `ln` in math.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network

- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.3 tensornn.loss.Loss

class tensornn.loss.**Loss**

Bases: ABC

Base loss class. All loss classes should inherit from this.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

abstract calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.4 tensornn.loss.MAE

class tensornn.loss.**MAE**

Bases: *Loss*

Mean absolute error is MSE but instead of squaring the values, you absolute value them.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.5 tensornn.loss.MSE

class tensornn.loss.**MSE**

Bases: *Loss*

Mean squared error is calculated extremely simply. 1. Find the difference between the prediction vs. the actual results we should have got 2. Square these values, because negatives are the same as positives, only magnitude matters 3. calculate mean

ex: our predictions: `[0.1, 0.2, 0.7]`, desired: `[0, 0, 1]` 1. pred - actual: `[0.1, 0.2, -0.3]` ` 2. squared: ``[0.01, 0.04, 0.09]` 3. mean: `0.04666667`

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the derivative of the loss function wrt the last layer of the network

1.4.6 tensornn.loss.MSLE

class tensornn.loss.**MSLE**

Bases: *Loss*

Mean squared logarithmic error is MSE but taking the log of our values before subtraction.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.7 tensornn.loss.Poisson

class tensornn.loss.**Poisson**

Bases: *Loss*

Poisson loss is calculated with this formula: average of (pred-desired*log(pred))

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.8 tensornn.loss.RMSE

class tensornn.loss.**RMSE**

Bases: *Loss*

Root mean squared error is just MSE, but it includes a square root after taking the average.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns
the derivative of the loss function wrt the last layer of the network

1.4.9 tensornn.loss.RSS

class tensornn.loss.**RSS**

Bases: *Loss*

Residual sum of squares loss is MSE but instead of doing the mean, you do the sum.

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the derivative of the loss function wrt the last layer of the network

1.4.10 tensornn.loss.SquaredHinge

class tensornn.loss.**SquaredHinge**

Bases: *Loss*

Square hinge loss is calculated with this formula: max(0, 1-pred*desired)^2

Methods

calculate	The mean of all the loss values in this batch.
derivative	Used in backpropagation which helps calculates how much each neuron impacts the loss.

calculate(*pred:* Tensor, *desired:* Tensor) → *Tensor*

The mean of all the loss values in this batch.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the average of calculated loss for one whole pass of the network

derivative(*pred:* Tensor, *desired:* Tensor) → *Tensor*

Used in backpropagation which helps calculates how much each neuron impacts the loss.

Parameters

- **pred** – the prediction of the network
- **desired** – the desired values which the network should have gotten close to

Returns

the derivative of the loss function wrt the last layer of the network

1.5 tensornn.nn

This file contains the neural network class.

Classes

NeuralNetwork	Create your neural network with this class.

1.5.1 tensornn.nn.NeuralNetwork

class tensornn.nn.**NeuralNetwork**(*layers: Iterable[*Layer*] = ()*)

Bases: object

Create your neural network with this class.

Methods

add	Add another layer(s) to the network.
backward	Find out how much each weight/bias contributes to the loss and gets stored in each layer.
forward	Propagate the inputs through the network and return the last layer's output.
get_loss	Calculate the loss for the given data.
predict	Get the prediction of the neural network.
register	Register the neural network.
simple	Create a NeuralNetwork from the number of neurons per layer.
train	Train the neural network.

__init__(*layers: Iterable[*Layer*] = ()*) → None

Initialize the network.

Parameters
layers – list of layers that make up network

__repr__()

Return repr(self).

add(*layers: Union[*Layer*, Iterable[*Layer*]]*) → None

Add another layer(s) to the network. This is the same as initializing the network with this layer included.

Parameters
layer – the layer to be added

backward(*loss_deriv:* Tensor) → None

Find out how much each weight/bias contributes to the loss and gets stored in each layer. Not meant for external use.

Parameters
loss_deriv – the derivative of the loss wrt the output of the last layer

Returns
nothing

forward(*inputs:* Tensor) → *Tensor*

Propagate the inputs through the network and return the last layer's output. Automatically flattens the input.

Parameters
inputs – inputs to the network

Returns
the output of the last layer in the network

get_loss(*inputs:* Tensor, *desired_outputs:* Tensor) → *Tensor*

Calculate the loss for the given data.

Parameters

- **inputs** – input to the network
- **desired_outputs** – desired output of the network for the given inputs

Returns
the loss of the network for the given parameters

predict(*inputs:* Tensor) → int

Get the prediction of the neural network. This will return the index of the most highly activated neuron. This method should only be used on a trained network, because otherwise it will produce useless random values.

Parameters
inputs – inputs to the network

Returns
an array which contains the value of each neuron in the last layer

register(*loss:* Loss, *optimizer:* Optimizer) → None

Register the neural network. This method initializes the network with loss and optimizer and also finishes up any last touches to its layers.

Parameters

- **loss** – type of loss this network uses to calculate loss
- **optimizer** – type of optimizer this network uses

Raises
InitializationError – num_inputs not specified to first layer

classmethod simple(*sizes: Sequence[int]*, *learning_rate: float = 0.001*)

Create a NeuralNetwork from the number of neurons per layer. First layer will be considered the input layer. All layers will be the Dense layer with the ReLU activation. The last layer will be Dense with the Softmax activation. The network will also be registered with CategoricalCrossEntropy loss and the SGD optimizer.

Parameters
sizes – list of numbers of neurons per layer

train(*inputs:* Tensor, *desired_outputs:* Tensor, *learning_rate: Optional[float] = None*, *batch_size: int = 32*, *epochs: int = 5*, *verbose: int = 1*) → None

Train the neural network. What training essentially does is adjust the weights and biases of the neural network for the inputs to match the desired outputs as close as possible.

Parameters

- **inputs** – training data which is inputted to the network
- **desired_outputs** – these values is what you want the network to output for respective inputs
- **epochs** – how many iterations will your network will run to learn
- **verbose** – the level of verbosity of the program (1-3), defaults to 1

Raises

- *NotRegisteredError* – network not registered
- *InputDimError* – inputs not at least 2d

1.6 tensornn.optimizers

This file contains optimizers that help tune your neural network. Optimizers enable us to improve our neural network efficiently.

Classes

Optimizer	
SGD	Stochastic gradient descent optimizer.

1.6.1 tensornn.optimizers.Optimizer

class tensornn.optimizers.**Optimizer**

Bases: ABC

Methods

1.6.2 tensornn.optimizers.SGD

class tensornn.optimizers.**SGD**(*learning_rate: float = 0.01*)

Bases: *Optimizer*

Stochastic gradient descent optimizer. But, this is actually mini-batch stochastic gradient descent. You can make it standard SGD by setting batch_size to 1 in training.

Methods

__init__(*learning_rate: float = 0.01*) → None

Initialize the optimizer.

Parameters

learning_rate – the learning rate of the optimizer

1.7 tensornn.tensor

This file contains the Tensor class for TensorNN. Essentially, it is just a class which does matrix operations for N-dimensional arrays. Currently we extend off of numpy's ndarray class since it is efficient and has a bunch of useful operations. If needed, we can add additional functionality to our extended class such as new methods.

Classes

Tensor	The tensor class.

1.7.1 tensornn.tensor.Tensor

class tensornn.tensor.**Tensor**(**args*, ***kwargs*)

Bases: ndarray

The tensor class. Currently functions like numpy.ndarray but with a custom print.

Methods

all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of *a* evaluate to True.
argmax	Return indices of the maximum values along the given axis.
argmin	Return indices of the minimum values along the given axis.
argpartition	Returns the indices that would partition this array.
argsort	Returns the indices that would sort this array.
astype	Copy of the array, cast to a specified type.
byteswap	Swap the bytes of the array elements
choose	Use an index array to construct a new array from a set of choices.
clip	Return an array whose values are limited to `[min, max]`.
compress	Return selected slices of this array along given axis.
conj	Complex-conjugate all elements.
conjugate	Return the complex conjugate, element-wise.

continues on next page

Table 1 – continued from previous page

copy	Return a copy of the array.
cumprod	Return the cumulative product of the elements along the given axis.
cumsum	Return the cumulative sum of the elements along the given axis.
diagonal	Return specified diagonals.
dot	Dot product of two arrays.
dump	Dump a pickle of the array to the specified file.
dumps	Returns the pickle of the array as a string.
fill	Fill the array with a scalar value.
flatten	Return a copy of the array collapsed into one dimension.
getfield	Returns a field of the given array as a certain type.
item	Copy an element of an array to a standard Python scalar and return it.
itemset	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max	Return the maximum along a given axis.
mean	Returns the average of the array elements along given axis.
min	Return the minimum along a given axis.
newbyteorder	Return the array with the same data viewed with a different byte order.
nonzero	Return the indices of the elements that are non-zero.
partition	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod	Return the product of the array elements over the given axis
ptp	Peak to peak (maximum - minimum) value along a given axis.
put	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel	Return a flattened array.
repeat	Repeat elements of an array.
reshape	Returns an array containing the same data with a new shape.
resize	Change shape and size of array in-place.
round	Return *a* with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of v should be inserted in a to maintain order.
setfield	Put a value into a specified place in a field defined by a data-type.
setflags	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort	Sort an array in-place.
squeeze	Remove axes of length one from *a*.
std	Returns the standard deviation of the array elements along given axis.

continues on next page

Table 1 – continued from previous page

sum	Return the sum of the array elements over the given axis.
swapaxes	Return a view of the array with *axis1* and *axis2* interchanged.
take	Return an array formed from the elements of *a* at the given indices.
tmp_str	Return str(self).
tobytes	Construct Python bytes containing the raw data bytes in the array.
tofile	Write array to a file as text or binary (default).
tolist	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring	A compatibility alias for *tobytes*, with exactly the same behavior.
trace	Return the sum along diagonals of the array.
transpose	Returns a view of the array with axes transposed.
var	Returns the variance of the array elements, along given axis.
view	New view of array with the same data.

Attributes

T	The transposed array.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

T

The transposed array.

Same as `self.transpose()`.

Examples

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

See Also

transpose

__abs__()

abs(self)

__add__(*value*, /)

Return self+value.

__and__(*value*, /)

Return self&value.

__array__([*dtype*], /) → reference if type unchanged, copy otherwise.

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

__array_finalize__

None.

__array_interface__

Array protocol: Python side.

__array_prepare__(*obj*) → Object of same type as ndarray object obj.

__array_priority__

Array priority.

__array_struct__

Array protocol: C-struct side.

__array_wrap__(*obj*) → Object of same type as ndarray object a.

__bool__()

self != 0

__contains__(*key*, /)

Return key in self.

__copy__()
Used if copy.copy() is called on an array. Returns a copy of the array.

Equivalent to a.copy(order='K').

__deepcopy__(*memo*, /) → Deep copy of array.
Used if copy.deepcopy() is called on an array.

__delitem__(*key*, /)
Delete self[key].

__divmod__(*value*, /)
Return divmod(self, value).

__eq__(*value*, /)
Return self==value.

__float__()
float(self)

__floordiv__(*value*, /)
Return self//value.

__format__()
Default object formatter.

__ge__(*value*, /)
Return self>=value.

__getitem__(*key*, /)
Return self[key].

__gt__(*value*, /)
Return self>value.

__hash__ = None

__iadd__(*value*, /)
Return self+=value.

__iand__(*value*, /)
Return self&=value.

__ifloordiv__(*value*, /)
Return self//=value.

__ilshift__(*value*, /)
Return self<<=value.

__imatmul__(*value*, /)
Return self@=value.

__imod__(*value*, /)
Return self%=value.

__imul__(*value*, /)
Return self*=value.

__index__()
Return self converted to an integer, if self is suitable for use as an index into a list.

__int__()
int(self)

__invert__()
~self

__ior__(*value*, /)
Return self|=value.

__ipow__(*value*, /)
Return self**=value.

__irshift__(*value*, /)
Return self>>=value.

__isub__(*value*, /)
Return self-=value.

__iter__()
Implement iter(self).

__itruediv__(*value*, /)
Return self/=value.

__ixor__(*value*, /)
Return self^=value.

__le__(*value*, /)
Return self<=value.

__len__()
Return len(self).

__lshift__(*value*, /)
Return self<<value.

__lt__(*value*, /)
Return self<value.

__matmul__(*value*, /)
Return self@value.

__mod__(*value*, /)
Return self%value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__neg__()
-self

static __new__(*cls*, **args*, ***kwargs*)

__or__(*value*, /)
Return self|value.

__pos__()
+self

__pow__(*value*, *mod=None*, /)
Return pow(self, value, mod).

__radd__(*value*, /)
Return value+self.

__rand__(*value*, /)
Return value&self.

__rdivmod__(*value*, /)
Return divmod(value, self).

__reduce__()
For pickling.

__reduce_ex__()
Helper for pickle.

__repr__()
Return repr(self).

__rfloordiv__(*value*, /)
Return value//self.

__rlshift__(*value*, /)
Return value<<self.

__rmatmul__(*value*, /)
Return value@self.

__rmod__(*value*, /)
Return value%self.

__rmul__(*value*, /)
Return value*self.

__ror__(*value*, /)
Return value|self.

__rpow__(*value*, *mod=None*, /)
Return pow(value, self, mod).

__rrshift__(*value*, /)
Return value>>self.

__rshift__(*value*, /)
Return self>>value.

__rsub__(*value*, /)
Return value-self.

__rtruediv__(*value*, /)
Return value/self.

__rxor__(*value*, /)
Return value^self.

__setitem__(*key*, *value*, /)
Set self[key] to value.

__setstate__(*state*, /)
For unpickling.

The *state* argument must be a sequence that contains the following elements:

Parameters

version
[int] optional pickle version. If omitted defaults to 0.

shape : tuple dtype : data-type isFortran : bool rawdata : string or list

a binary string with the data (or a list if 'a' is an object array)

__sizeof__()
Size of object in memory, in bytes.

__str__()
Return str(self).

__sub__(*value*, /)
Return self-value.

__truediv__(*value*, /)
Return self/value.

__xor__(*value*, /)
Return self^value.

all(*axis=None*, *out=None*, *keepdims=False*, *, *where=True*)
Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

numpy.all : equivalent function

any(*axis=None*, *out=None*, *keepdims=False*, *, *where=True*)
Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See Also

numpy.any : equivalent function

argmax(*axis=None*, *out=None*)

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

See Also

numpy.argmax : equivalent function

argmin(*axis=None*, *out=None*)

Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

See Also

numpy.argmin : equivalent function

argpartition(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

New in version 1.8.0.

See Also

numpy.argpartition : equivalent function

argsort(*axis=-1*, *kind=None*, *order=None*)

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

See Also

numpy.argsort : equivalent function

astype(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

Parameters

dtype
[str or dtype] Typecode or data-type to which the array is cast.

order
[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting
[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] 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.

subok
[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy
[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t
[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning
When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

base

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

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

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

byteswap(*inplace=False*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace
[bool, optional] If `True`, swap bytes in-place, default is `False`.

Returns

out
: [ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

`A.newbyteorder().byteswap()` produces an array with the same values
: but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

`choose`(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

numpy.choose : equivalent function

`clip`(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *numpy.clip* for full documentation.

See Also

numpy.clip : equivalent function

`compress`(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

See Also

numpy.compress : equivalent function

`conj`()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

See Also

numpy.conjugate : equivalent function

`conjugate`()

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

numpy.conjugate : equivalent function

`copy`(*order='C'*)

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

See also

numpy.copy : Similar function with different default behavior numpy.copyto

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

ctypes

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters

None

Returns

c

[Python object] Possessing attributes data, shape, strides, etc.

See Also

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.**data**

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.**shape**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be *ctypes.c_int*, *ctypes.c_long*, or *ctypes.c_longlong* depending on the platform. The c_intp type is defined accordingly in *numpy.ctypeslib*. The ctypes array contains the shape of the underlying array.

_ctypes.**strides**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

_ctypes.**data_as**(*obj*)

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

_ctypes.**shape_as**(*obj*)

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

_ctypes.**strides_as**(*obj*)

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

cumprod(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

See Also

numpy.cumprod : equivalent function

cumsum(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

See Also

numpy.cumsum : equivalent function

data

Python buffer object pointing to the start of the array's data.

diagonal(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See Also

numpy.diagonal : equivalent function

dot(*b*, *out=None*)

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

See Also

numpy.dot : equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

dtype

Data-type of the array's elements.

Parameters

None

Returns

d : numpy dtype object

See Also

numpy.dtype

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

dump(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file
: [str or Path] A string naming the dump file.

  Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

fill(*value*)

Fill the array with a scalar value.

Parameters

value
: [scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)
The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)
The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)
The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)
The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)
The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)
This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

UPDATEIFCOPY (U)
(Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC
F_CONTIGUOUS and not C_CONTIGUOUS.

FORC
F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)
ALIGNED and WRITEABLE.

CARRAY (CA)
BEHAVED and C_CONTIGUOUS.

FARRAY (FA)
BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.

- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

flatten : Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order
[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y
[ndarray] A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array. flat : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype
[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset
[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

item(**args*)

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

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size* == *1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z
: [Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

`itemset`(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args**
: [Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

itemsize

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

max(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

numpy.amax : equivalent function

mean(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *, *where=True*)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See Also

numpy.mean : equivalent function

min(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See Also

numpy.amin : equivalent function

nbytes

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

newbyteorder(*new_order='S', /*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

new_order
[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian
- '=' - native order, equivalent to *sys.byteorder*
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr
[array] New array object with the dtype reflecting given change to the byte order.

`nonzero()`

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

numpy.nonzero : equivalent function

`partition(kth, axis=-1, kind='introselect', order=None)`

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth
[int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis
[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind
[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order
[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be

specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

See Also

numpy.prod : equivalent function

ptp(*axis=None*, *out=None*, *keepdims=False*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

See Also

numpy.ptp : equivalent function

put(*indices*, *values*, *mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

numpy.put : equivalent function

ravel([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

real

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

numpy.real : equivalent function

repeat(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

numpy.repeat : equivalent function

reshape(*shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

numpy.reshape : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

`resize`(*new_shape*, *refcheck=True*)

Change shape and size of array in-place.

Parameters

new_shape
[tuple of ints, or *n* ints] Shape of resized array.

refcheck
[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError
If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError
If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

resize : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See Also

numpy.around : equivalent function

searchsorted(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

See Also

numpy.searchsorted : equivalent function

setfield(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val
: [object] Value to be placed in field.

dtype
: [dtype object] Data-type of the field in which to place *val*.

offset
: [int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
```

(continues on next page)

(continued from previous page)

```
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write
: [bool, optional] Describes whether or not *a* can be written to.

align
: [bool, optional] Describes whether or not *a* is aligned properly for its type.

uic
: [bool, optional] Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

(continues on next page)

(continued from previous page)

```
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

numpy.reshape : similar function ndarray.reshape : similar method

size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

sort(*axis=-1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis
: [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind
: [{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

  Changed in version 1.15.0: The 'stable' option was added.

order
: [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be

specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

numpy.sort : Return a sorted copy of an array. numpy.argsort : Indirect sort. numpy.lexsort : Indirect stable sort on multiple keys. numpy.searchsorted : Find elements in sorted array. numpy.partition: Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze(*axis=None*)

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

See Also

numpy.squeeze : equivalent function

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See Also

numpy.std : equivalent function

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

See Also

numpy.lib.stride_tricks.as_strided

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
```

(continues on next page)

(continued from previous page)

```
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)

Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

See Also

numpy.sum : equivalent function

swapaxes(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See Also

numpy.swapaxes : equivalent function

take(*indices*, *axis=None*, *out=None*, *mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

See Also

numpy.take : equivalent function

tmp_str()

Return str(self).

tobytes(*order='C'*)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order
: [{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s
[bytes] Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

`tofile`(*fid*, *sep=''*, *format='%s'*)

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters

fid
[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep
[str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format
[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

`tolist`()

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *~numpy.ndarray.item* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

y
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring(*order='C'*)

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

numpy.trace : equivalent function

transpose(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out
: [ndarray] View of *a*, with axes suitably permuted.

See Also

transpose : Equivalent function ndarray.T : Array property returning the array transposed. ndarray.reshape : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
```

(continues on next page)

(continued from previous page)

```
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)

Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

See Also

numpy.var : equivalent function

view(*[dtype][, type]*)

New view of array with the same data.

Note: Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

dtype
: [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type
: [Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
```

(continues on next page)

(continued from previous page)

```
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

1.8 tensornn.utils

This file contains useful variables that are used in TensorNN.

Functions

normalize	Normalize the training data.
one_hot	Get the one-hot representation of an integer.
source	Get the source code of a TensorNN object.
takes_one_hot	Apply this decorator to a function that takes in a one-hot vector.
takes_single_value	Apply this decorator to a function that takes in a single value.

1.8.1 tensornn.utils.normalize

tensornn.utils.**normalize**(*data*)

Normalize the training data. This will never hurt your data, it will always help it, make sure to use this every time. This function will make it so that the largest value in the data is 1.

Parameters
data – training data to the network

Returns
normalized data, max is 1

1.8.2 tensornn.utils.one_hot

tensornn.utils.**one_hot**(*values: Union[int, Iterable[int]]*, *classes: int*) → *Tensor*

Get the one-hot representation of an integer. One-hot representation is like the opposite of np.argmax. Let's we want our network's output to be [0, 1](first neuron on, second off), that would be the 'one-hot vector'. If you were to run np.argmax([0, 1]), you would get the index of the 1(which is also the index of the max value).

Parameters

- **values** – to be converted to one-hot (max 1D), ex: one_hot(3, 5) -> [0, 0, 0, 1, 0]
- **classes** – number of different places for the 1, len of one-hot

Returns
one-hot vector from the given params

1.8.3 tensornn.utils.source

tensornn.utils.**source**(*obj: ~typing.Any*, *output: ~typing.Optional[~typing.TextIO] = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → str

Get the source code of a TensorNN object.

Parameters
obj – the tensornn object, ex: tnn.nn.NeuralNetwork

1.8.4 tensornn.utils.takes_one_hot

tensornn.utils.**takes_one_hot**(*pos: int = 2*)

Apply this decorator to a function that takes in a one-hot vector. Used for loss functions.

Parameters
pos – position of the argument to convert to one-hot vector, default 2 for loss functions

Returns
decorator

1.8.5 tensornn.utils.takes_single_value

tensornn.utils.**takes_single_value**(*pos: int = 1*)

Apply this decorator to a function that takes in a single value. Used for loss functions.

Parameters
pos – position of the argument to convert to single value, default 1 for loss functions

Returns
decorator

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V