Cheatography

Data Science In Go: A Cheat Sheet

by chewxy via cheatography.com/42427/cs/12774/

Introduction

Go is the future for doing data science. In this cheatsheet, we look at 2 families of libraries that will allow you to do that.

They are: `gorgon ia.o rg /tensor` and `gonum.o rg /v1 /g o num/mat` . The gonum libraries will be referred to as `gonum/mat`

For this cheatsheet, assume the following:

```
import ts " gor gon ia.o rg /te nso r"
```

Note on panic and error behaviour.

1. Most `tensor` operations return `error`.
2. `gonum` has a good policy of when errors are returned and when panics happen.

What To Use

I ever only want a `float64` matrix or vector

> use `gonum/mat`

I want to focus on doing statistical/scientific work

> use `gonum/mat`

I want to focus on doing machine learning work

> use `gonum/mat` or `gorgon ia.o rg /tensor`.

I want to focus on deep learning work

> use `gorgon ia.o rg /tensor`

I want multidimensional arrays

> use `gorgon ia.o rg /tensor`, or `[]mat.M atrix`

I want to work with different data types

> use `gorgon ia.o rg /tensor`

What To Use (cont)

I want to wrangle data like in Pandas or R - with data frames

> use `kniren /gota`

Default Values

Numpy

```
a = np.Zer os( (2,3))
```

gonum/mat

```
a := mat.Ne wDe nse(2, 3, nil)
```

tensor

```
a := ts.New (ts.Of (Fl oat32), ts.Wit hSh ape (2,3))
```

A Range...

Numpy

```
a = np.ara nge(0, 9).res hap e(3,3)
```

gonum

```
a := mat.NewDense(3, 3, floats.Span(make([]float64, 
```

tensor

```
a := ts.New (ts.Wi thB ack ing (ts.Ra nge (ts.Int, 0,
ape (3,3))
```

Identity Matrices

Numpy

```
a = np.eye (3,3)
```

gonum/mat

```
a := mat.Ne wDi ago nal(3, []floa t64{1, 1, 1})
```

tensor

```
a := ts.I(3, 3, 0)
```

Elementwise Arithmetic Operations

Addition

Numpy

```
c = a + b
c = np.add(a, b)
a += b                          # in-place
np.add(a, b, out=c) # reuse array
```

gonum/mat

```
c.Add(a, b)
a.Add(a, b) // in-place
```

By **chewxy**
cheatography.com/chewxy/

Published 12th October, 2017.
Last updated 25th October, 2020.

Elementwise Arithmetic Operations (cont)

tensor

```
var c *ts.Dense; c, err = a.Add(b)
var c ts.Tensor; c, err = ts.Add(a, b)
a.Add(b, ts.Use Uns afe())          // in-place
a.Add(b, ts.Wit hRe use(c))        // reuse tensor
ts.Add(a, b, ts.Use Uns afe())  // in-place
ts.Add(a, b, ts.Wit hRe use(c)) // reuse
```

Note: The operations all returns a result and an error, omitted for brevity here. It's good habit to check for errors.

Subtraction

Numpy

```
c = a - b
c = np.sub tra ct(a, b)
```

gonum/mat

```
c.Sub(a, b)
```

tensor

```
c, err:= a.Sub(b)
c, err = ts.Sub(a, b)
```

Multiplication

Numpy

```
c = a * b
c = np.mul tip ly(a, b)
```

gonum/mat

```
c.MulE lem(a, b)
```

tensor

```
c, err := a.Mul(b)
c, err := ts.Mul(a, b)
```

Division

Numpy

```
c = a / b
c = np.div ide(a, b)
```

gonum/mat

```
c.DivE lem(a, b)
```

Elementwise Arithmetic Operations (cont)

tensor

```
c, err := a.Div(b)
c, err := ts.Div(a, b)
```

Note: When encountering division by 0 for non-floats, an error will be returned, and the value at which the offending value will be 0 in the result.

Note: All variations of arithmetic operations follow the patterns available in *Addition* for all examples.

Note on Shapes

In all of these functions, `a` and `b` has to be of the same shape. In Numpy operations with dissimilar shapes will throw an exception. With `gonum/mat` it'd panic. With `tensor`, it will be returned as an error.

Aggregation

Sum

Numpy

```
s = a.sum()
s = np.sum(a)
```

gonum/mat

```
var s float64 = mat.Sum(a)
```

tensor

```
var s *ts.Dense = a.Sum()
var s ts.Tensor = ts.Sum(a)
```

Note: The result, which is a scalar value in this case, can be retrieved by calling `s.Scal arV alue()`

Sum Along An Axis

Numpy

```
s = a.sum( axis=0)
s = np.sum(a, axis=0)
```

gonum/mat

Write a loop, with manual aid from `mat.Col` and `mat.Row`

Note: There's no performance loss by writing a loop. In fact there arguably may be a cognitive gain in being aware of what one is doing.

Aggregation (cont)

tensor

```
var s *ts.Dense = a.Sum(0)
var s ts.Tensor = ts.Sum(a, 0)
```

Argmax/Argmin

Numpy

```
am = a.argmax()
am = np.arg max(a)
```

gonum

Write a loop, using `mat.Col` and `mat.Row`

tensor

```
var am *ts.Dense; am, err = a.Argmax(ts.AllAxes)
var am ts.Tensor; am, err = ts.Argmax(a, ts.AllAxes)
```

Argmax/Argmin Along An Axis

Numpy

```
am = a.argm ax( axis=0)
am = np.arg max(a, axis=0)
```

gonum

Write a loop, using `mat.Col` and `mat.Row`

tensor

```
var am *ts.Dense; am, err = a.Argm ax(0)
var am ts.Tensor; am, err = ts.Arg max(a, 0)
```

Data Structure Creation

Numpy

```
a = np.arr ay([1, 2, 3])
```

gonum/mat

```
a := mat.Ne wDe nse(1, 3, []floa t64{1, 2, 3})
```

tensor

```
a := ts.New (ts.Wi thB ack ing ([] int{1, 2, 3})
```

Creating a float64 matrix

Numpy

```
a = np.arr ay([[0, 1, 2], [3, 4, 5]], dtype= 'fl oat64')
```

gonum/mat

```
a := mat.Ne wDe nse(2, 3, []floa t64{0, 1, 2, 3, 4, 5})
```

tensor

```
a := ts.New (ts.Wi thB ack ing ([] flo at64{0, 1, 2, 3, 4, 5}, ts.Wit hSh ap
e(2, 3))
```

Creating a float32 3-D array

Data Structure Creation (cont)

Numpy

```
a = np.arr ay( [[[0, 1, 2], [3, 4, 5]], [[100, 101, 1
```

tensor

```
a := ts.New(ts.WithShape(2, 2, 3), ts.WithBacking([]
```

Note: The the `tensor` package is imported as `ts`

Additionally, `gonum/mat` actually offers many different data structures, each being useful to a particular subset of computations. The examples given in this document mainly assumes a dense matrix.

gonum Types

`mat.Matrix`	Abstract data type representing any `float64` matrix
`*mat.Dense`	Data type representing a dense `float64` matrix

tensor Types

`tensor.Tenso r`	An abstract data type representing any kind of tensors. Package functions work on these types.
`*tensor.Dens e`	A representation of a densely packed multidimensional array. Methods return `*tenso r.Dense` instead of `tensor.Tensor`

By **chewxy**
cheatography.com/chewxy/

Published 12th October, 2017.
Last updated 25th October, 2020.

tensor Types (cont)

`*tensor.CS`	A representation of compressed sparse row/column matrices.
`*tensor.MA`	*Coming soon* - representation of masked multidimensional array. Methods return `*tensor.MA` instead of `tensor.Tensor`
`tensor.DenseTensor`	Utility type that represents densely packed multidimensional arrays
`tensor.MaskedTensor`	Utility type that represents densely packed multidimensional arrays that are masked by a slice of `bool`
`tensor.Sparse`	Utility type that represents any sparsely packed multi-dim arrays (for now: only `*CS`)

Metadata

Metadata	Numpy	gonum	tensor
Shape	`a.shape`	`a.Dims()`	`a.Shape()`
Strides	`a.strides`		`a.Strides()`
Dims	`a.ndim`		`a.Dims()`

Tensor Manipulation

Zero-op Transpose

Numpy

```
aT = a.T
```

gonum/mat

```
aT := a.T()
```

tensor

```
a.T()
```

Transpose With Data Movement

Numpy

```
aT = np.tra nsp ose(a)
```

gonum/mat

```
b := a.T(); aT := mat.De nse Cop yOf(b)
```

tensor

```
aT, err := ts.Tra nsp ose(a)
```

or

```
a.T(); err := a.Tran spose()
```

Reshape

Numpy

```
b = a.resh ape (2,3)
```

gonum/mat

```
b := NewDen se(2, 3, a.RawM atr ix( ).Data)
```

tensor

```
err := a.Resh ape (2,3)
```

Note on reshaping when using gonum: the matrix `a` mustn't be a view.

Linear Algebra

Inner Product of Vectors

Numpy

```
c = np.inn er(a, b)
```

gonum

```
var c float64 = mat.Dot(a, b)
```

tensor

```
var c interf ace{} = ts.Inn er(a, b)
```

or

```
var c interf ace{} = a.Inner(b)
```

Note: The `tensor` package comes with specialized execution engines for `float64` and `float32` which will return `float64` or `float32` without returning an `interf ace{}`

Matrix-Vector Multiplication

Linear Algebra (cont)

Numpy

```
mv = np.dot(m, v)
```

or

```
mv = np.mat mul(m, v)
```

or

```
mv = m @ v
```

or

```
mv = m.dot(v)
```

gonum

```
mv.Mul(m, v)
```

tensor

```
var mv ts.Tensor; mv, _ = ts.Mat Vec Mul(m, v)
```

or

```
var mv *Dense; mv, _ = m.MatV ecM ul(v)
```

Matrix-Matrix Multiplication

Numpy

```
mm = np.dot(m1, m2)
```

or

```
mm = np.mat mul(m1, m2)
```

or

```
mm = m1 @ m2
```

or

```
mm = m1.dot(m2)
```

gonum

```
mm.Mul(m1, m2)
```

tensor

```
var mm Tensor; mm, _ = ts.Mat Mul(m1, m2)
```

or

```
var mm *ts.Dense; mm, _ = m1.Mat Mul(m2)
```

Magic

Numpy

```
c = np.dot(a, b)
c = a.dot(b)
```

tensor

```
var c ts.Tensor; c, _ = ts.Dot(a, b)
var c *ts.Dense; c, _ = a.Dot(b)
```

Note: The `Dot` function and method in package `tensor` works similarly to `dot` in Numpy - depending on the number of dimensions of the inputs, different functions will be called. You should treat it as a "magic" function that does products of two multi-dimensional arrays.

`gonum` has a whole suite of linear-algebra functions and structures that are too many to enumerate here. You should check it out too.

Combinations

Concatenation

Numpy

```
c = np.con cat ena te((a, b), axis=0)
```

gonum/mat

```
c.Stac k(a,b)
```

tensor

```
var c ts.Tensor; c, err = ts.Con cat(0, a, b)
var c *ts.Dense; c, err = a.Conc at(0, b)
```

Vstack

Numpy

```
c = np.vst ack((a, b))
```

gonum/mat

```
c.Stac k(a,b)
```

tensor

```
var c *ts.Dense; c, err = a.Vsta ck(0, b)
```

Hstack

Numpy

```
c = np.hst ack((a, b))
```

gonum/mat

```
c.Augm ent (a,b)
```

tensor

```
var c *ts.Dense; c, err = a.Hsta ck(0, b)
```

Stack onto a New Axis

Numpy

```
c = np.sta ck((a, b))
```

gonum/mat

```
var stacked []mat.M atrix; stacked = append (st acke
 b)
```

tensor

```
var c ts.Tensor; c, _ = ts.Sta ck(0, a, b)
var c *ts.Dense; c, _ = a.Stac k(0,b)
```

Note: Unlike in Numpy, `Stack` in `tensor` is a little more strict on the a
has to be specified.

Repeats

Combinations (cont)

Numpy

```
c = np.rep eat(a, 2) # returns a flat array
c = np.rep eat(a, 2, axis=0) # repeats along axis 0
c = np.rep eat(a, 2, axis=1) # repeats along axis 1
```

gonum

Unsupported for now

tensor

```
var c ts.Tensor; c, _ = ts.Rep eat(a, ts.All Axes,  2) // returns a flat arra
y
c = ts.Rep eat(a, 0, 2) // repeats along axis 0
c = ts.Rep eat(a, 1, 2) // repeats along axis 1
```

Data Access

Value At (Assuming Matrices)

Numpy

```
val = a[0, 0]
```

gonum/mat

```
var va float64 := a.At(0, 0)
```

tensor

```
var val interf ace{}; val, _ = a.At(0,0)
```

Slice Row or Column (Assuming Matrices)

Numpy

```
row = a[0]
col = a[:, 0]
```

gonum/mat

```
var row mat.Vector = a.RowV iew(0)
var col mat.Vector = a.ColV iew(0)
```

tensor

```
var row ts.View = a.Slic e(s(0))
var col ts.View = a.Slic e(nil, s(0))
```

Advanced Slicing (Assuming 9x9 Matrices)

Numpy

```
b = a[1:4, 3:6]
```

gonum/mat

```
var b mat.Matrix = a.Slic e(1,4, 3,6)
```

tensor

```
var b ts.View = a.Slic e(r s(1,4), rs(3,6))
```

Advanced Slicing With Steps

Data Access (cont)

Numpy

```
b = a[1:4:1, 3:6:2]
```

gonum/mat

Unsupported

tensor

```
var b ts.View = a.Slic e(r s(1 ,4,1), rs(3,6,2))
```

Getting Underlying Data

Numpy

```
b = a.ravel()
```

gonum/mat

```
var b []float64 = a.RawM atr ix( ).Data
```

tensor

```
var b interf ace{} = a.Data()
```

Setting One Value (Assuming Matrices)

Numpy

```
a[r, c] = 100
```

gonum/mat

```
a.Set(r, c, 100)
```

tensor

```
a.SetA t(100, r, c)
```

Setting Row/Col (Assuming 3x3 Matrix)

Numpy

```
a[r] = [1, 2, 3]
a[:, c] = [1, 2, 3]
```

gonum/mat

```
a.SetR ow(r, []floa t64{1, 2, 3})
a.SetC ol(c, []floa t64{1, 2, 3})
```

tensor

No simple method - requires `Iterator`s and multiple lines of code.

Note: in the `tensor` examples, the `a.Slice` method take a list of `t ensor.Slice` which is an interface defined here. `s`, and `rs` in the examples simply represent types that implement the `tensor.Slice` type. A `nil` is treated as a `:` in Python. There are no default `tenso r.Slice` types provided, and it is up to the user to define their own.