stem Documentation

Release 0.0.2

Abraham Schneider

Jun 10, 2017

Contents

1 Tensor Overview **3**

1.1 Synopsis . . . 3
1.2 Storage . . . 4
1.3 Tensor Class . . . 5
1.4 Tensor Construction . . . 5
1.5 Indexing . . . 6
1.6 Views . . . 6

2 Tensor Functions **7**

2.1 Basic Math . . . 7
2.2 Linear Algerbra . . . 9
2.3 Tensor Manipulation . . . 9
2.4 Statistics . . . 10
2.5 Other . . . 10

3 Random library **11**

4 Neural Network library **13**

4.1 Overview . . . 13
4.2 The Op class . . . 15

5 Indices and tables **19**

6 Status **21**

Contents:

CHAPTER 1

Tensor Overview

Synopsis

First, we will define a convenience alias:

```
typealias NativeStorage<Double> D
```

Next, we can create two vectors:

```
// construct vector with values
let v1 = Tensor<D>([1, 2, 3])

// construct Tensor with a specific size
let v2 = Tensor<D>(Extent(5, 3))

// construct a matrix with values
let v3 = Tensor<D>([[1, 2, 3], [4, 5, 6]])
```

STEM supports standard linear algebra operators:

```
// take the dot product (result is a scalar)
let s1 = v1v2

// take the outer product (result is a matrix)
let m1 = v1v2

// add two vectors together
let v4 = v1+v3

// multiply by a scalar
let v5 = 0.5*v1
```

STEM also supports advanced indexing (similar to Numpy and Matlab):

```
let v6 = v2[1..<4]
let m2 = m1[1..<4, 0..<2]
```

As **STEM**'s name implies N-dimensional Tensors are supported. Both the `Vector` and `Matrix` classes are specializations of the `Tensor` class. These specializations allow for simpler construction methods as well as the use of accelerated libraries such as **CBLAS** and **CUDA** or **OpenCL** through function overloading.

Storage

All `Tensor` s have an associated `Storage` class that is responsible for the allocated memory. The two built-in `Storage` types are: `NativeStorage` and `CBlasStorage`. Other storage types (e.g. **CUDA** or **OpenCL**) can be added without requiring any rewrite of the main library. Because the `Storage` type determines which functions get called. If no methods have been specified for the `Storage` class, `NativeStorage` will be called by default.

The `Storage` protocol is defined as:

```
public protocol Storage {
  associatedtype ElementType:NumericType

  var size:Int { get }
  var order:DimensionOrder { get }

  init(size:Int)
  init(array:[ElementType])
  init(storage:Self)
  init(storage:Self, copy:Bool)

  subscript(index:Int) -> ElementType {get set}

  // returns the order of dimensions to traverse
  func calculateOrder(dims:Int) -> [Int]

  // re-order list in order of dimensions to traverse
  func calculateOrder(values:[Int]) -> [Int]
}
```

An implementation of `Storage` determines the allocation through the `init` methods, `subscript` determines how the storage gets indexed, and `calculateStride` allows the `Storage` to be iterated through in a sequential fashion.

The `Tensor` class frequently makes use of the generator `IndexGenerator` to iterate through the `Storage` class. This provides a convenient way to access all the elements without knowing the underyling memory allocation.

To do so, the `Tensor` class defined the method:

```
public func indices(order:DimensionOrder?=nil) -> GeneratorSequence<IndexGenerator> {
  if let o = order {
      return GeneratorSequence<IndexGenerator>(IndexGenerator(shape, order: o))
  } else {
      return GeneratorSequence<IndexGenerator>(IndexGenerator(shape, order: storage.
↪order))
  }
}
```

which can be used like:

```
func fill<StorageType:Storage>(tensor:Tensor<StorageType>, value:StorageType.
↪ElementType) {
    for i in tensor.indices() {
        tensor.storage[i] = value
    }
}
```

However, as mentioned previously, if an optimized version for a particular `Tensor` operation exists, you can write:

```
// This will be used if the Tensor's storage type is CBlasStorage for doubles,
// an alternative can be specified for Floats separately.
func fill(tensor:Tensor<CBlasStorage<Double>>, value:StorageType.ElementType) {
  // call custom library
}
```

Table 1.1: Storage Types

Type	Description
NativeStorage	Unaccelerated using row-major memory storage
CBlasStorage	CBLAS acceleration using column-major storage
GPUStorage	(Not Implemented) GPU acceleration using row-memory storage

Tensor Class

The `Tensor` class is parameterized by the `Storage` type, allowing instances of the class to maintain a pointer to the underlying memory. The `Tensor` class also has an instance of `ViewType`, which allows different views of the same memory to be constructed, and the array `dimIndex`, which determines the order that the dimensions in the `Tensor` are traversed. These features allow for multiple `Tensor` s to provide a different view to the same memory (e.g. a slice of a `Tensor` can be created by changing the `ViewType` instance, or a `Tensor` can be transposed by shuffling `dimIndex`).

Note: Throughout the documentation `Tensor<S>` indicates the parameterization of the `Tensor` class by `Storage` type `S`, and `NumericType` refers to `S.NumericType` (see section on `Storage` for details).

Tensor Construction

`Tensor<S>(_ shape:Extent)`
Constructs a tensor with the given shape.

`Tensor<S>([NumericType], axis:Int)`
Constructs a vector along the given axis

`Tensor<S>(colvector:[NumericType])`
Constructs a column vector (equivalent to `Tensor<S>([NumericType], axis:0)`)

`Tensor<S>(rowvector:[NumericType])`
Constructs a row vector (equivalent to `Tensor<S>([NumericType], axis:1)`)

`Tensor<S>([[NumericType]])`
Constructs a matrix

Indexing

STEM supports single indexing as well as slice indexing. Given a `Tensor` T:

To index element (i, j, k):

```
let value = T[i, j, k]
T[i, j, k] = value
```

To index the slices (if:il, jf:jl, kf:kl):

```
let T2 = T[if...il, jf...jl, kf...kl]
T[if...il, jf...jl, kf...kl] = T2
```

Views

Views in **STEM** are instances of `Tensor` that point to the same `Storage` as another `Tensor` but with different bounds and/or ordering of dimensions. Views are most commonly created whenever a slice indexing is used.

A copy of a view can be made by using the `copy` function.

CHAPTER 2

Tensor Functions

Basic Math

add (*lhs:Tensor<S>*, *rhs:Tensor<S>*, *result:Tensor<S>*)
add (*lhs:Tensor<S>*, *rhs:NumericType*, *result:Tensor<S>*)
add (*lhs:NumericType*, *rhs:Tensor<S>*, *result:Tensor<S>*)
+(lhs:Tensor<S>, rhs:Tensor<S>) -> Tensor<S>
+(lhs:Tensor<S>, rhs:NumericType) -> Tensor<S>
+(lhs:NumericType, rhs:Tensor<S>) -> Tensor<S>

Adds `lhs` and `rhs` together, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

sub (*lhs:Tensor<S>*, *rhs:Tensor<S>*, *result:Tensor<S>*)
sub (*lhs:Tensor<S>*, *rhs:NumericType*, *result:Tensor<S>*)
sub (*lhs:NumericType*, *rhs:Tensor<S>*, *result:Tensor<S>*)
-(lhs:Tensor<S>, rhs:Tensor<S>) -> Tensor<S>
-(lhs:Tensor<S>, rhs:NumericType) -> Tensor<S>
-(lhs:NumericType, rhs:Tensor<S>) -> Tensor<S>

Subtracts `lhs` from `rhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

mul (*lhs:Tensor<S>*, *rhs:Tensor<S>*, *result:Tensor<S>*)
mul (*lhs:Tensor<S>*, *rhs:NumericType*, *result:Tensor<S>*)
mul (*lhs:NumericType*, *rhs:Tensor<S>*, *result:Tensor<S>*)
***(lhs:Tensor<S>, rhs:Tensor<S>) -> Tensor<S>**
***(lhs:Tensor<S>, rhs:NumericType) -> Tensor<S>**
***(lhs:NumericType, rhs:Tensor<S>) -> Tensor<S>**

Performs element-wise multiplication between `lhs` and `rhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

div (*lhs:Tensor<S>*, *rhs:Tensor<S>*, *result:Tensor<S>*)

`div` (*lhs:Tensor<S>*, *rhs:NumericType*, *result:Tensor<S>*)
`div` (*lhs:NumericType*, *rhs:Tensor<S>*, *result:Tensor<S>*)
`/(lhs:Tensor<S>, rhs:Tensor<S>) -> Tensor<S>`
`/(lhs:Tensor<S>, rhs:NumericType) -> Tensor<S>`
`/(lhs:NumericType, rhs:Tensor<S>) -> Tensor<S>`

Performs element-wise division between `lhs` and `rhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

`iadd` (*lhs:Tensor<S>*, *rhs:Tensor<S>*)
`iadd` (*lhs:Tensor<S>*, *rhs:NumericType*)
`iadd` (*lhs:NumericType*, *rhs:Tensor<S>*)
`+=(lhs:Tensor<S>, rhs:Tensor<S>)`
`+=(lhs:Tensor<S>, rhs:NumericType)`
`+=(lhs:NumericType, rhs:Tensor<S>)`

Adds `lhs` to `rhs` and stores result in `lhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

`isub` (*lhs:Tensor<S>*, *rhs:Tensor<S>*)
`isub` (*lhs:Tensor<S>*, *rhs:NumericType*)
`isub` (*lhs:NumericType*, *rhs:Tensor<S>*)
`-=(lhs:Tensor<S>, rhs:Tensor<S>)`
`-=(lhs:Tensor<S>, rhs:NumericType)`
`-=(lhs:NumericType, rhs:Tensor<S>)`

Subtracts `rhs` from `lhs` and stores result in `lhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

`imul` (*lhs:Tensor<S>*, *rhs:Tensor<S>*)
`imul` (*lhs:Tensor<S>*, *rhs:NumericType*)
`imul` (*lhs:NumericType*, *rhs:Tensor<S>*)
`*=(lhs:Tensor<S>, rhs:Tensor<S>)`
`*=(lhs:Tensor<S>, rhs:NumericType)`
`*=(lhs:NumericType, rhs:Tensor<S>)`

Multiplies `lhs` by `rhs` and stores result in `lhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

`idiv` (*lhs:Tensor<S>*, *rhs:Tensor<S>*)
`idiv` (*lhs:Tensor<S>*, *rhs:NumericType*)
`idiv` (*lhs:NumericType*, *rhs:Tensor<S>*)
`/=(lhs:Tensor<S>, rhs:Tensor<S>)`
`/=(lhs:Tensor<S>, rhs:NumericType)`
`/=(lhs:NumericType, rhs:Tensor<S>)`

Divides `lhs` by `rhs` and stores results in `lhs`, either place results in `result` or returning its value. If the shape of `lhs` and `rhs` do not match, will either use `broadcast` to match their size, or use optimized methods to accomplish the same purpose.

pow (*lhs:Tensor<S>*, *exp:NumericType*) → Tensor<S>
`**(lhs:Tensor<S>, exp:NumericType) -> Tensor<S>`

Raises every element of `lhs` by `exp`.

exp (*op::Tensor<S>*) → Tensor<S>

Returns an element-wise exponentiation of `op`.

abs (*op:Tensor<S>*) → Tensor<S>

Performs an elementwise `abs` on Tensor.

sum (*op:Tensor<S>*, *axis:Int*) → Tensor<S>
sum (*op:Tensor<S>*) → NumericType
When `axis` is specified, sums `op` along `axis` and returns resulting Tensor. When no `axis` is specified, returns entire Tensor summed.

max (*op:Tensor<S>*, *axis:Int*) → Tensor<S>
max (*op:Tensor<S>*) → NumericType
When `axis` is specified, find the maximum value `op` across `axis` and returns resulting Tensor. When no `axis` is specified, returns maximum value of entire Tensor.

min (*op:Tensor<S>*, *axis:Int*) → Tensor<S>
min (*op:Tensor<S>*) → NumericType
When `axis` is specified, find the minimum value `op` across `axis` and returns resulting Tensor. When no `axis` is specified, returns minimum value of entire Tensor.

Linear Algerbra

dot (*lhs:Tensor<S>*, *rhs:Tensor<S>*) → NumericType
(lhs:Tensor<S>, rhs:Tensor<S>) -> NumericType
Returns the dot product between two vectors. Both `lhs` and `rhs` must be vectors.

outer (*lhs:Tensor<S>*, *rhs:Tensor<S>*) → NumericType
(lhs:Tensor<S>, rhs:Tensor<S>) -> NumericType
Returns the outer product between two vectors. Both `lhs` and `rhs` must be vectors.

Tensor Manipulation

fill (*op:Tensor<S>*, *value:NumericType*)
Sets all elements of `op` to `value`.

copy (*op:Tensor<S>*) → Tensor<S>
copy (*from:Tensor<S>*, *to:Tensor<S>*)
First form creates a new Tensor and copies `op` into it. The second form copies `op` into an already existing Tensor.

map (*op:Tensor<S>*, *fn:(NumericType) -> NumericType*) → Tensor<S>
Applies `fn` to each element of Tensor and returns resulting Tensor.

reduce (*op:Tensor<S>*, *fn:(Tensor<S>*, *Tensor<S>) -> NumericType*) → NumericType
Applies `fn` to elements of Tensor, returns a scalar.

concat (*op1:Tensor<S>*, *op2:Tensor<S>*, *...*, *opN:Tensor<S>*, *axis:Int*)
Concats N Tensors along `axis`.

vstack (*op1:Tensor<S>*, *op2:Tensor<S>*) → Tensor<S>
Returns a new Tensor that is a composite of `op1` and `op2` vertically stacked.

hstack (*op1:Tensor<S>*, *op2:Tensor<S>*) → Tensor<S>
Returns a new Tensor that is a composite of `op1` and `op2` horizontally stacked.

broadcast (*op1:Tensor<S>*, *op2:Tensor<S>) -> (Tensor<S>*, *Tensor<S>*)
Returns two Tensors of the same shape broadcasted from `op1` and `op2`. Need to go into much more detail about broadcasting.

Statistics

`norm`(*op:Tensor*, *axis:Int*) → Tensor<S>
    Calculates the norm along specified axis.

`hist`(*op:Tensor, bins:[Int]*) → Tensor<S>
    Returns a vector with size of `bins` with the resulting histogram of `op`.

Other

`isClose`(*lhs:Tensor<S>*, *rhs:Tensor<S>*, *eps:NumericType*) → Bool
    Returns if `lhs` is within the range of (`rhs` + `eps`, `rhs` - `eps`)

CHAPTER 3

Random library

To generate a random number you first need to create a *RandomNumberGenerator*:

```
let rng = RandomNumberGenerator(seed)
```

The *seed* variable is optional, and if left out will be set the system's clock.

The *random* library can be used in two ways:

1. Fill an already existing *Tensor* with random values
2. Create a new *Tensor* that is filled with random values

CHAPTER 4

Neural Network library

Overview

The basic building block is an `Op`, which is parameterized by the underyling storage type (see the `Tensor` documentation for more details). Every `Op` as the function `apply`, which performs a transform on the `Op` s input updating the *output* variable. `Op` s may have a variable number of inputs and outputs.

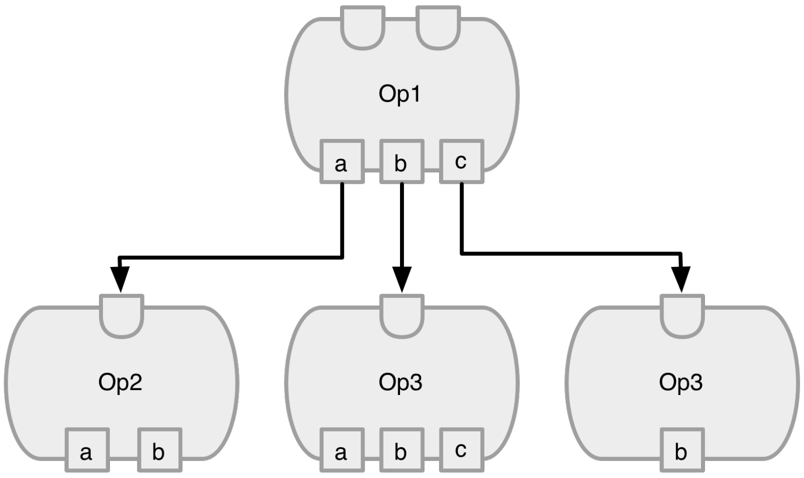

Fig. 4.1: **Figure 1**: A computation graph is formed by connecting multiple `Op` s together. The inputs are a collection of `Op` s, and the outputs, the result of the operation applied to the input, are a collection of `Tensor` s.

The design of the `Op` class is keep the details of how the optimization is performed separate from the transform itself. For example, the protocol `Differentiable` provides a method to extend the `Op` class with the `gradient` method for stochastic gradient descent optimization. The `gradient` methods returns a new `Op` whose transform is the derivative of the target `Op`. This allows a secondary computation graph to easily be constructed:

```
let linear = Linear<D>(inputSize: 5, outputSize: 3)
let linearGrad = linear.gradient()
```

Some `Op` s, like `Sequence` are collections of other `Op` s, and will invoke (via `apply`) each of the contained `Op` s in a pre-determined order. For example, a standard neural network can be created with:

```
let net = Sequence<D>(
  VariableOp<D>(zeros(Extent(5))),
  Linear<D>(inputSize: 5, outputSize: 3),
  Sigmoid<D>(size: 3),
  L2Loss<D>(size: 3))

net.apply()
```

There are a few important items of note in this example:

1. The `apply` method takes no parameters and has no return value
2. The input to the first layer of the network is of the type `Variable`, which is also an `Op`
3. The loss function `L2Loss` is also an `Op`

The design decision to make everything an `Op` allows the creation of the computational graph. In this case, all of the `Op` s are also `Differentiable`, and thus you can do:

```
let netGrad = net.gradient()
```

Optimization can be performed with the following code:

```
params = net.params()
gradParams = netGrad.params()

for i in 0..<iterations {
  netGrad.reset()

  net.apply()
  netGrad.apply()

  for (param, gradParam) in Zip2Sequences(params, gradParams) {
    param += 0.01*gradParam
  }
}
```

However, this code can be simplified using an `Optimizer`:

```
let alpha = VariableOp<D>(0.01)
let opt = GradientDescentOptimizer(net, alpha: alpha)

for i in 0..<iterations {
  opt.apply()
}
```

where `GradientDescentOptimizer` automatically constructs the gradient network and collects the parmaeters for both the forward and backward sequences.

One of the advantages to having everything be an operation in the computation graph is that the `alpha` variable can be set dynamically. For example, if a momentum optimization is desired, the `alpha` variable can be computed from the current error.

The Op class

The `Op` class has the following properties:

- id: unique ID for instance of `Op`
- inputs: collection of `Op` s
- output: collection `Tensor` s that are a result of the transform

and has the following methods defined:

- apply(): performs transform on inputs and stores results in `output`
- params(): returns all the parameters of the transform (e.g. if its a `Linear` Op, then the parameters are `weight` and `bias`).

OrderedDictionary

One important detail about the `inputs` and `outputs` of an `Op` is that it is maintained by the `OrderedDictionary` class. An instance of the `OrderedDictionary` class maintains a dictionary of (`String`: [`T`]), but also provides a method to traverse the items in the order that they were added. This provides a guarantee in the order of traversal as well as provide a method for efficient access (e.g. if an `Op` has a specific ordering of `inputs`, an integer index may be used instead of a `String`).

By maintaining an array of `T` means that a single entry in the `OrderedDictionary` may be a collection of items. This provides an easy way to create `Op` s that have a variable number of inputs and/or outputs. For example, the `AddOp` can take in `N` inputs and will provide a single output.

Op library

`Linear``()`
Performs a linear transformation on input.

`Sigmoid``()`
Applies the sigmoid function to each element of the input.

`Tanh``()`
Applies the Tanh function to each element of the input

`AddOp``()`
Adds a collection of inputs together

`MulOp``()`
Multiplies a collection of inputs together

`Concat``()`
Concatenates a series of inputs together

`L2Loss``()`
Takes two inputs: `value` and `target`. Calculates the square distance between the two.

Creating a new Op

Suppose you wanted to create an `Op` that takes the log of the input. The `Log` op can be defined as:

```
public class Log<S:Storage where S.ElementType:FloatNumericType>: Op<S> {
  public init(size:Int) {
    super.init( inputs: [NoOp<S>()],
                output: Tensor<S>(Extent(size)),
                labels: ["input"])
  }

  public override func apply() {
    if output == nil || output!.shape != inputs[0].output!.shape {
      output = Tensor<S>(Extent(inputs[0].output!.shape))
    }

    log(inputs[0].output!, result: output!)
  }
}
```

where the initialization defines a single input (`input`) that is currently not defined (the `NoOp`) and the output is allocated as the size specified by the parameter. The `apply` function finds the maximum value in the input, divides each element of the input by that value, and stores in the result in `output`.

The gradient of `Log` can be defined as:

The `Log` gradient takes two additional inputs: the instance of the `Log` op its the gradient of, and `gradOutput`, which is the gradient of the op's output.

Finally, to allow the gradient to be taken of `Log`, the class must be extended to `Differentiable`:

```
extension Log:Differentiable {
  public func gradient() -> GradientType {
    return LogGrad<S>(op: self)
  }
}
```

We can change the construction of our network by adding `Log` into the sequence:

```
let net = Sequence<D>(
  VariableOp<D>(zeros(Extent(5))),
  Log<D>(size: 5)
  Linear<D>(inputSize: 5, outputSize: 3),
  Sigmoid<D>(size: 3),
  L2Loss<D>(size: 3))
```

and have the optimization correctly calculate the derivative as before:

```
let opt = GradientDescentOptimizer(net, alpha: alpha)
```

because `GradientDescentOptimizer` will automatically call `gradient` on each `Op`, an instance of `LogGradient` will be created for each instance of `Log`.

Testing your Op

It is always a good idea to do a gradient check on a newly created `Op`. You can create a new unit test to do so:

```
func testLogOpGradient() {
  let eps = 10e-6
  let input = VariableOp<S>(uniform(Extent(10)))
  let gradOutput = VariableOp<S>(zeros(Extent(10)))
  let log = Log<S>(size: 10)
  log.setInput("input", to: input)

  let logGrad = log.gradient() as! LogGrad<S>
  logGrad.setInput("gradOutput", to: gradOutput)

  // test gradient wrt to the input
  let inputError = checkGradient(log, grad: logGrad, params: input.output,␣
↪gradParams: logGrad.output, eps: eps)
  XCTAssertLessThan(inputError, eps)
}
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Status

STEM is very much a work in progress. It should not be considered stable yet. The status will be kept updated for when the design of STEM has settled down.

If a Tensor library for Swift interests you, please help by contributing code.

Index

A

abs() (built-in function), 8
add() (built-in function), 7
AddOp() (built-in function), 15

B

broadcast() (built-in function), 9

C

Concat() (built-in function), 15
concat() (built-in function), 9
copy() (built-in function), 9

D

div() (built-in function), 7
dot() (built-in function), 9

E

exp() (built-in function), 8

F

fill() (built-in function), 9

H

hist() (built-in function), 10
hstack() (built-in function), 9

I

iadd() (built-in function), 8
idiv() (built-in function), 8
imul() (built-in function), 8
isClose() (built-in function), 10
isub() (built-in function), 8

L

L2Loss() (built-in function), 15
Linear() (built-in function), 15

M

map() (built-in function), 9
max() (built-in function), 9
min() (built-in function), 9
mul() (built-in function), 7
MulOp() (built-in function), 15

N

norm() (built-in function), 10

O

outer() (built-in function), 9

P

pow() (built-in function), 8

R

reduce() (built-in function), 9

S

Sigmoid() (built-in function), 15
sub() (built-in function), 7
sum() (built-in function), 9

T

Tanh() (built-in function), 15

V

vstack() (built-in function), 9