

Micrograd

A presentation by Andrej Karpathi on Aug 2022 in [The spelled-out intro to neural networks and backpropagation: building micrograd](#)

Links:

- [micrograd on github](#)
- [Graphviz documentation](#)
- [Julia graphviz](#)

Table of Contents

Micrograd
Data Structure
Visualization
Manual backpropagation and gradient
Backpropagation programmatically
Re-implementing tanh using basic building blocks
Julia Apparte - Conversion and Promotion rules

```
• begin
•   using PlutoUI
•   PlutoUI.TableOfContents(indent=true, depth=4, aside=true)
• end
```

Data Structure

```
• mutable struct Value{T <: Real}
•   data::T
•   _prev::Set
•   _op::Symbol
•   _backward::Function
•   label::String
•   grad::T
•
•   function Value{T}(data::T;
•       _children::Tuple=(),
•       _op::Symbol=:_,
•       label::String="",
•   ) where {T <: Real}
•       grad = zero(T)
•       _backward = () -> nothing # default to Nothing
•       new{T}(data, Set{_children}, _op, _backward, label, grad)
•   end
•
•   # Value{T}(data::S) where {T <: Real, S <: Integer} =
•   #   Value{T}(T(data))
• end
```

const DT = Float64

```
• const DT = Float64
```

YaValue (generic function with 1 method)

```
• # default constructor for Float64
• function YaValue(data::T; _children::Tuple=(), _op::Symbol=:_, label::String="") where {T <: Real}
•   Value{T}(data; _children, _op, label)
• end
```

```
• import Base: +, -, *, /
```

```
• function Base.:+(self::Value{T}, other::Value{T}) where {T <: Real}
•   y = YaValue(self.data + other.data; _children=(self, other), _op=:+)
•   function _backward_fn()
•       self.grad += 1.0 * y.grad
•       other.grad += 1.0 * y.grad
•   end
•   y._backward = _backward_fn
•   y
• end
```

```

• function Base.:+(self::Value{T}, other::Value{T}) where {T <: Real}
•     y = YaValue(self.data + other.data; _children=(self, other), _op=:+)
•     function _backward_fn()
•         self.grad += 1.0 * y.grad
•         other.grad += 1.0 * y.grad
•     end
•     y._backward = _backward_fn
•     y
• end

```

```

• function Base.:*(self::Value{T}, other::Value{T}) where {T <: Real}
•     y = YaValue(self.data * other.data; _children=(self, other), _op=:*)
•     function _backward_fn()
•         self.grad += other.data * y.grad
•         other.grad += self.data * y.grad
•     end
•     y._backward = _backward_fn
•     y
• end

```

```

• Base.show(io::IO, self::Value) = print(io, "Value(data=$(self.data))")

```

```

• function Base.tanh(self::Value{T}) where {T <: Real}
•     x = exp(2*self.data)
•     tanh = (x - 1.) / (x + 1.)
•     y = YaValue(tanh; _children=(self, ), _op=:tanh, label="tanh")
•     function _backward_fn()
•         self.grad += (1. - tanh^2) * y.grad
•     end
•     y._backward = _backward_fn
•     y
• end

```

backward (generic function with 1 method)

```

• function backward(self::Value{T}) where {T <: Real}
•     topo, visited = [], Set()
•     function build_topological_order(v::Value)
•         if v ∉ visited
•             push!(visited, v)
•             for child ∈ v._prev
•                 build_topological_order(child)
•             end
•             push!(topo, v)
•         end
•     end
•     self.grad = 1.0
•     for cnode ∈ build_topological_order(self) |> reverse
•         cnode._backward()
•     end
• end

```

"Output"

```

• begin
•     a = YaValue(2.0; label="a")
•     b = YaValue(-3.0; label="b")
•     c = YaValue(10.0; label="c")
•
•     d = a * b; d.label = "d"
•     e = d + c; e.label = "e"
•
•     f = YaValue(-2.0; label="f")
•     L = e * f; L.label="Output"
• end

```

```

(Set([Value(data=2.0), Value(data=-3.0)]), :*)

```

```

• d._prev, d._op

```

Visualization

```

• using GraphViz, FileIO, Cairo

```

```

• using Printf

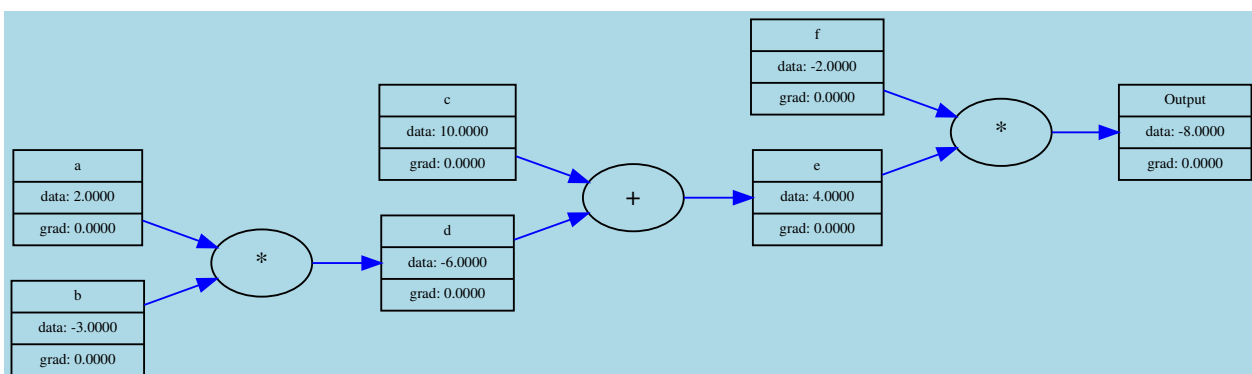
```

draw_dot (generic function with 1 method)

```

• begin
•   # for visualization
•
•   function trace(root::Value)
•       # builds a set of all nodes and all edges in a graph
•       nodes, edges = Set(), Set()
•       function build(v::Value)
•           if v ∉ nodes
•               push!(nodes, v)
•               for child ∈ v._prev
•                   push!(edges, (child, v))
•                   build(child)
•               end
•           end
•       end
•       build(root)
•       nodes, edges
•   end
•
•   function draw_dot(root::Value)
•       gr = ""
•       format=svg;
•       rankdir="LR";
•       dpi=72;
•       bgcolor=lightblue;
•       imagepos="mc";
•       landscape=false;
•       mode="hier";
•       layout=dot
•       node [shape=record];
•       "" # Left to Right
•       nodes, edges = trace(root)
•       for n ∈ nodes
•           uid = string(objectid(n))
•           gr = string(gr,
•               ""
•               $(uid) [name=$(uid),label="$(Printf.@sprintf "%s | data: %.4f | grad: %.4f" n.label n.data
•                   n.grad)",fontsize=8];
•               ""
•           )
•           if n._op != :_
•               gr = string(gr,
•                   ""
•                   "$(string(uid, n._op))" [name=$(string(uid)),label="$(string(n._op))",shape="ellipse"];
•                   "$(string(uid, n._op))" -- $(uid) [color=blue,dir=forward];
•                   ""
•               )
•           end
•       end
•       for (n1, n2) ∈ edges
•           gr = string(gr,
•               ""
•               $(string(objectid(n1))) -- "$(string(objectid(n2), n2._op))" [color=blue,dir=forward];
•               ""
•           )
•       end
•       gr = string("""graph G {""", gr, ""}""")
•       # dot""
•       # $(gr)
•       # ""
•       open("digraph.dot", "w") do io
•           write(io, gr)
•       end
•
•       open("digraph.dot", "r") do io
•           GraphViz.load(io)
•       end
•   end
• end

```



• draw_dot(L)

Manual backpropagation and gradient

try_grad (generic function with 1 method)

```

• function try_grad()
•   h = 0.001
•
•   a = YaValue(2.0; label="a")
•   b = YaValue(-3.0; label="b")
•   c = YaValue(10.0; label="c")
•   f = YaValue(-2.0; label="f")
•   # compose
•   d = a * b; d.label = "d"
•   e = d + c; e.label = "e"
•   L = e * f; L.label="Output"
•
•   a1 = YaValue(a.data; label="a")
•   b1 = YaValue(b.data; label="b")
•   c1 = YaValue(c.data; label="c")
•   f1 = YaValue(f.data; label="f")
•   # compose
•   d1 = a1 * b1; d1.label = "d"
•   d1.data += h
•   e1 = d1 + c1; e1.label = "e"
•   L1 = e1 * f1; L1.label="Output"
•
•   Δh = (L1.data - L.data) / h
• end

```

-2.000000000000668

```

• # got 7 var =>
• try_grad()

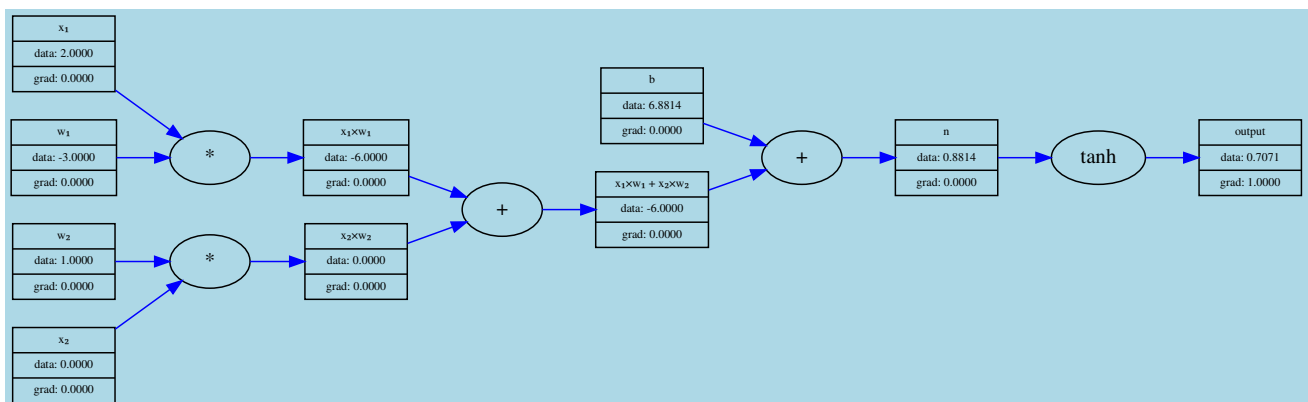
```

one_neuron (generic function with 1 method)

```

• function one_neuron()
•   # 2 inputs
•   x1, x2 = YaValue(2.0; label="x1"), YaValue(0.0; label="x2")
•   # 2 weights
•   w1, w2 = YaValue(-3.0; label="w1"), YaValue(1.0; label="w2")
•   # bias
•   b = YaValue(6.8813735870195432; label="b")
•
•   x1w1 = x1 * w1
•   x1w1.label = "x1×w1"
•   x2w2 = x2 * w2
•   x2w2.label = "x2×w2"
•
•   # x1w1 + x2w2 + b
•   x1w1x2w2 = x1w1 + x2w2
•   x1w1x2w2.label = "x1×w1 + x2×w2"
•   n = x1w1x2w2 + b
•   n.label = "n"
•
•   o = tanh(n)
•   o.label = "output"
•   (o, n, x1w1x2w2, b, x1w1, x2w2, x1, x2, w1, w2)
• end

```



```

• begin
•   (o, n, x1w1x2w2, bias, x1w1, x2w2, x1, x2, w1, w2) = one_neuron()
•   o.grad = 1.0
•   draw_dot(o)
• end

```

Let's do backpropagation through tanh. So what is $\frac{do}{dn}$ given $o = \tanh(n)$?

By definition: $\frac{do}{dn} = 1 - o^2 = 1 - \tanh(n)^2$

(0.5, 0.5)

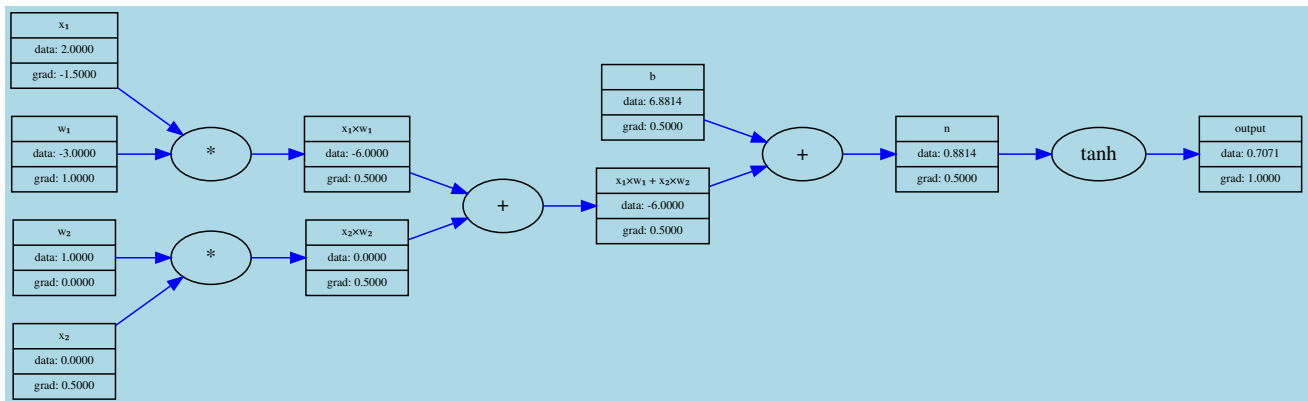
```
• begin
•   n.grad = 1 - o.data^2
•
•   # we also can fill in the gradient for  $x_1w_1x_2w_2$ ,  $b$  - applying + rule
•    $x_1w_1x_2w_2$ .grad = bias.grad = n.grad
•
•   # and for  $x_1w_1$ ,  $x_2w_2$  - applying + rule
•    $x_1w_1$ .grad,  $x_2w_2$ .grad =  $x_1w_1x_2w_2$ .grad,  $x_1w_1x_2w_2$ .grad
• end
```

(0.5, 0.0)

```
•  $x_2$ .grad,  $w_2$ .grad =  $w_2$ .data *  $x_2w_2$ .grad,  $x_2$ .data *  $x_2w_2$ .grad
```

(-1.5, 1.0)

```
•  $x_1$ .grad,  $w_1$ .grad =  $w_1$ .data *  $x_1w_1$ .grad,  $x_1$ .data *  $x_1w_1$ .grad
```



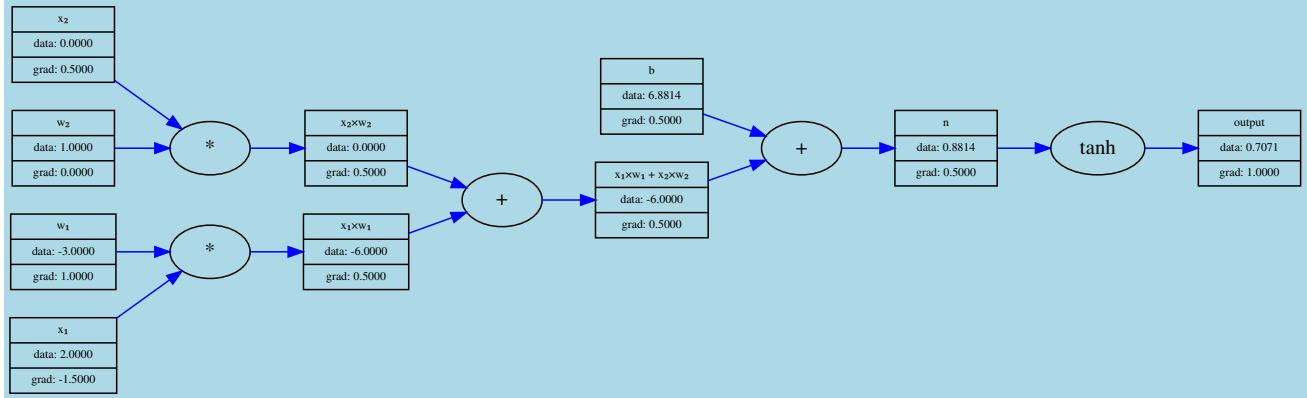
```
• # redraw graph with gradient updated
• draw_dot(o)
```

Backpropagation programmatically

Of course, we need to update all arithmetic operations on our datatype.

backprop_one_neuron (generic function with 1 method)

```
• function backprop_one_neuron()
•   # 2 inputs
•    $x_1$ ,  $x_2$  = YaValue(2.0; label=" $x_1$ "), YaValue(0.0; label=" $x_2$ ")
•   # 2 weights
•    $w_1$ ,  $w_2$  = YaValue(-3.0; label=" $w_1$ "), YaValue(1.0; label=" $w_2$ ")
•   # bias
•    $b$  = YaValue(6.8813735870195432; label=" $b$ ")
•
•    $x_1w_1 = x_1 * w_1$ 
•    $x_1w_1$ .label = " $x_1 \times w_1$ "
•    $x_2w_2 = x_2 * w_2$ 
•    $x_2w_2$ .label = " $x_2 \times w_2$ "
•
•   #  $x_1w_1 + x_2w_2 + b$ 
•    $x_1w_1x_2w_2 = x_1w_1 + x_2w_2$ 
•    $x_1w_1x_2w_2$ .label = " $x_1 \times w_1 + x_2 \times w_2$ "
•    $n = x_1w_1x_2w_2 + b$ 
•    $n$ .label = " $n$ "
•
•    $o = \tanh(n)$ 
•    $o$ .label = "output"
•   # ( $o$ ,  $n$ ,  $x_1w_1x_2w_2$ ,  $b$ ,  $x_1w_1$ ,  $x_2w_2$ ,  $x_1$ ,  $x_2$ ,  $w_1$ ,  $w_2$ )
•
•   # and now the backward pass
•    $o$ .grad = 1.0
•    $o$ ._backward()
•    $n$ ._backward()
•    $x_1w_1x_2w_2$ ._backward()
•    $x_1w_1$ ._backward()
•    $x_2w_2$ ._backward()
•   #  $b$ ._backward()
•    $o$ 
• end
```



```

• begin
•   o1 = backprop_one_neuron()
•   draw_dot(o1)
• end

```

Note: we can backpropagate given an order: the reverse of a topological order of the graph...

topological_order (generic function with 1 method)

```

• function topological_order(o::Value)
•   topo, visited = [], Set()
•   function build_topological_order(v::Value)
•     if v ∉ visited
•       push!(visited, v)
•       for child ∈ v._prev
•         build_topological_order(child)
•       end
•       push!(topo, v)
•     end
•   end
•   build_topological_order(o)
• end

```

[Value(data=-3.0), Value(data=2.0), Value(data=-6.0), Value(data=0.0), Value(data=1.0), Value(data=0.0), Value(data=-6.0),

```

• topological_order(o1)

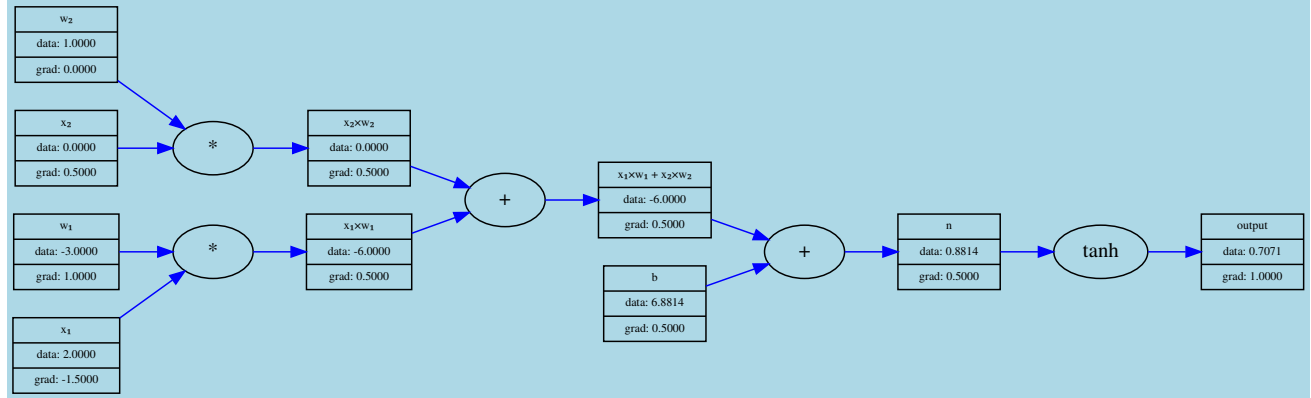
```

auto_backprop_one_neuron (generic function with 1 method)

```

• function auto_backprop_one_neuron()
•   # 2 inputs
•   x1, x2 = YaValue(2.0; label="x1"), YaValue(0.0; label="x2")
•   # 2 weights
•   w1, w2 = YaValue(-3.0; label="w1"), YaValue(1.0; label="w2")
•   # bias
•   b = YaValue(6.8813735870195432; label="b")
•
•   # forward pass
•   x1w1 = x1 * w1
•   x1w1.label = "x1*w1"
•   x2w2 = x2 * w2
•   x2w2.label = "x2*w2"
•
•   # x1w1 + x2w2 + b
•   x1w1x2w2 = x1w1 + x2w2
•   x1w1x2w2.label = "x1*w1 + x2*w2"
•   n = x1w1x2w2 + b
•   n.label = "n"
•
•   o = tanh(n)
•   o.label = "output"
•
•   # and now the backward pass, using reverse order of the graph's topological order
•   o.grad = 1.0
•   for cnode ∈ topological_order(o) |> reverse
•     cnode._backward()
•   end
•   o
• end

```



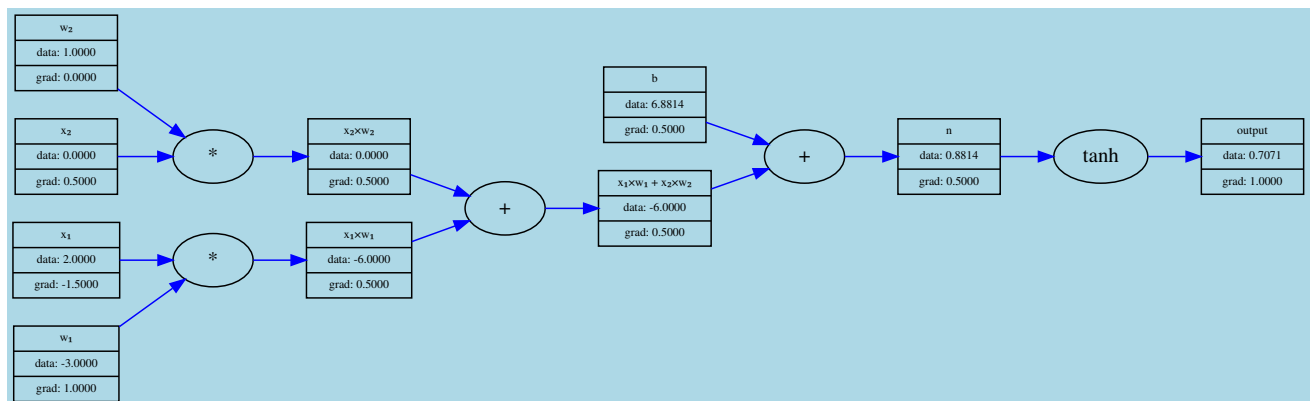
```

• begin
•   o2 = auto_backward_one_neuron()
•   draw_dot(o2)
• end

```

After defining the function `backward` on our datatype (`Value`) we can invoke it!

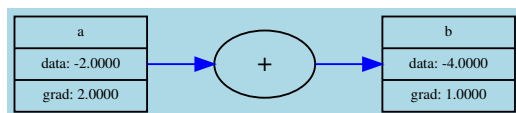
Let's do this...



```

• begin
•   (o3, _rest_) = one_neuron()
•   backward(o3)
•   draw_dot(o3)
• end

```



```

• begin
•   # need to use += in _backward() function on Value type.
•   aa = YaValue(-2.0, label="a")
•   bb = aa + aa
•   bb.label = "b"
•   backward(bb)
•   draw_dot(bb) # double arrow from a to :+ - expected ∇(aa) == 2.
• end

```

Re-implementing tanh using basic building blocks

Julia Apparte - Conversion and Promotion rules

First we want to be able to write something like

```

a = Value(2.0, label="a")
a + 1 # MethodError: no method matching +(...)

```

As it is with our datatype, this is not working because `1` is not a `Value` it is just an integer. OK, so let's add some methods (in Julia terminology) for our arithmetic operators, namely by adding promotion rules.

promote_rule (generic function with 147 methods)

```
• begin
•   import Base: promote_rule, convert
•
•   # these two allow: promote(xr, r) where xr is Value{Float64} and r is Float64 => Value{Float64}
•   #                       promote(xi, i) where xi is Value{Int64} and i is Int64 => Value{Int64}
•   convert(::Type{Value{T}}, x::T) where {T <: Real} = Value{T}(x)
•   promote_rule(::Type{Value{T}}, ::Type{T}) where {T <: Real} = Value{T}
•
•   # Value{Float64} and Float32 => Value{Float64}
•   convert(::Type{Value{T}}, x::S) where {T <: Real, S <: AbstractFloat} = Value{T}(T(x))
•   promote_rule(::Type{Value{T}}, ::Type{S}) where {T <: Real, S <: AbstractFloat} = Value{T}
•
•   # Value{Float64} and Integer => Value{Float64}
•   convert(::Type{Value{T}}, x::S) where {T <: Real, S <: Integer} = Value{T}(T(x))
•   promote_rule(::Type{Value{T}}, ::Type{S}) where {T <: Real, S <: Integer} = Value{T}
•
•   convert(::Type{Value{T}}, x::Type{Value{S}}) where {T <: Real, S <: T} = Value{T}(T(x.data))
•   promote_rule(::Type{Value{T}}, ::Type{Value{S}}) where {T <: Real, S <: T} = Value{promote_type(T, S)}
• end
```

(Value{Float64}, Value{Float32}, Value{Int64}, Value{Int32})

```
• begin
•   vf64 = YaValue(2.0, label="vf64")
•   vf32 = YaValue(Float32(2.0), label="vf32")
•   vi64 = YaValue(2, label="vi64")
•   vi32 = YaValue(Int32(2), label="vi32")
•
•   typeof(vf64), typeof(vf32), typeof(vi64), typeof(vi32)
• end
```

((Value(data=2), Value(data=4)), (Value(data=2), Value(data=16)))

```
• begin
•   # from Int -> Value{Int}
•   i64, i32 = 4, Int32(16)
•   promote(vi64, i64), promote(vi64, i32)
• end
```

(Value(data=2.0), Value(data=2.0))

```
• begin
•   # from Float -> Value{Float}
•   f64 = 2.0
•   promote(vf64, f64)
• end
```

(Value(data=2.0), Value(data=3.1415927410125732))

```
• begin
•   # from Float32 -> Value{Float64}
•   #      Float16 -> Value{Float32} ...
•   f32 = Float32(π)
•   promote(vf64, f32)
• end
```

(Value(data=2.0), Value(data=16.0))

```
• # from Int -> Value{Float}
• promote(vf64, i32)
```

```
• ## Allowing Value{T} + T => Value{T}
• Base.:+(self::Value{T}, other::T) where {T <: Real} = +(self, Value{T}(other))
```

(Value(data=6), Value(data=18), Value{Int64}, Value{Int64})

```
• vi64 + i64, vi64 + i32, typeof(vi64 + i64), typeof(vi64 + i32)
```

```
• # Allowing Value{T} + S => Value{T} where S <: T
• Base.:+(self::Value{T}, other::S) where {T <: Real, S <: Integer} =
•   +(self, Value{T}(T(other)))
```

(Value(data=6.0), Value(data=18.0), Value{Float64}, Value{Float64})

```
• vf64 + i64, vf64 + i32, typeof(vf64 + i64), typeof(vf64 + i32)
```

(Value(data=4.0), Value{Float64}, Float64)

```
• vf64 + f64, typeof(vf64), typeof(f64)
```

```
• # Allowing Value{T} + Value{S} => Value{T} where S <: T
• Base.:+(self::Value{T}, other::Value{S}) where {T <: Real, S <: Real} =
•   +(self, Value{T}(T(other.data)))
```

(Value(data=4.0), Value{Float64}, Value{Int32})

```
• # Value{Float64} + Value{Int64}
• vf64 + vi64, typeof(vf64), typeof(vi32)
```


myadd (generic function with 1 method)

```
• function myadd(self::Value{T}, other::Value{S}) where {T <: AbstractFloat, S <: T}
•   ST = supertype(T) # supertype.((T, S))[1]
•   +(Value{ST}(ST(self.data)), Value{ST}(ST(other.data)))
• end
• # Causing issues => xa + xi: stackoverflow, xa + rx: stackoverflow, xa + i: stackoverflow
```

(Value{data=4.0}, Value{Float64}, Value{Float32})

```
• # Value{Float64} + Value{Float32}
• vf64 + vf32, typeof(vf64), typeof(vf32)
```

(AbstractFloat, AbstractFloat)

```
• supertype.((Float32, Float64))
```

subtypetree (generic function with 3 methods)

```
• function subtypetree(rtype, level=1, indent=2)
•   level == 1 && (println(rtype))
•   for st ∈ subtypes(rtype)
•     println(string(repeat(" ", level * indent), st))
•     subtypetree(st, level + 1, indent)
•   end
• end
```

```
• subtypetree(Real)
```

```
Real
  AbstractFloat
    BigFloat
    Float16
    Float32
    Float64
  AbstractIrrational
    Irrational
  FixedPointNumbers.FixedPoint
  FixedPointNumbers.Fixed
  FixedPointNumbers.Normed
  Integer
  Bool
  Signed
    BigInt
    Int128
    Int16
    Int32
    Int64
    Int8
  Unsigned
    UInt128
    UInt16
    UInt32
    UInt64
    UInt8
  Rational
```

```
• subtypetree(Integer)
```

```
Integer
  Bool
  Signed
    BigInt
    Int128
    Int16
    Int32
    Int64
    Int8
  Unsigned
    UInt128
    UInt16
    UInt32
    UInt64
    UInt8
```

```
• subtypetree(AbstractFloat)
```

```
AbstractFloat
  BigFloat
  Float16
  Float32
  Float64
```

```
• subtypetree(Value{Real})
```

```
Main.var"workspace#5".Value{Real}
```