Ctrl + S

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


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

Data Structure

```julia
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
end
```

```julia
import Base: +, -, *, /, ^
```

const DT = Float64

```julia
const DT = Float64
```

YaValue (generic function with 1 method)

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

```julia
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
```

```julia
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
```

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

```julia
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)

```julia
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"

```julia
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(2.0, Set([]), :_, #2, "a", 0.0), Value(-3.0, Set([]), :_, #2, "b", 0.0)]), :*)

```julia
d._prev, d._op
```

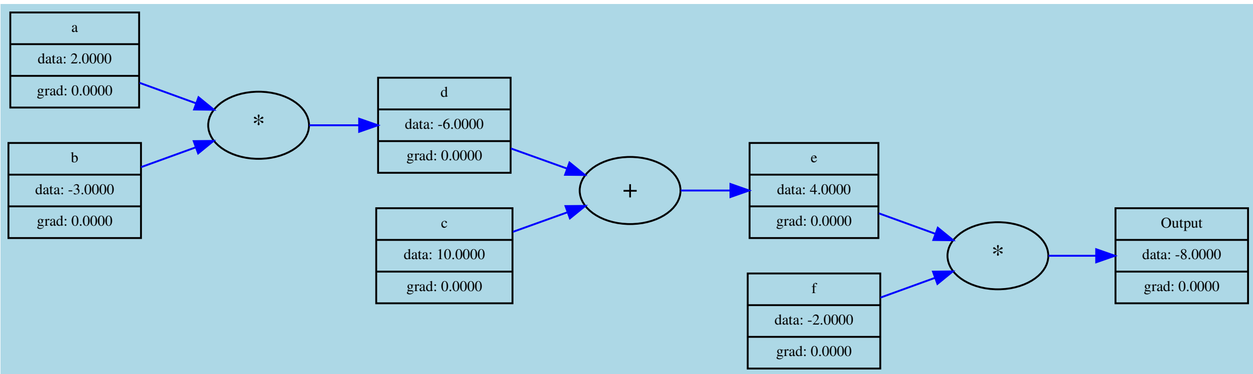
Visualization

```julia
using GraphViz    , FileIO    , Cairo
```

```julia
using Printf
```

draw_dot (generic function with 1 method)

```julia
## for visualization
begin
    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
```




```julia
draw_dot(L)
```

Manual backpropagation and gradient

```
try_grad (generic function with 1 method)
```

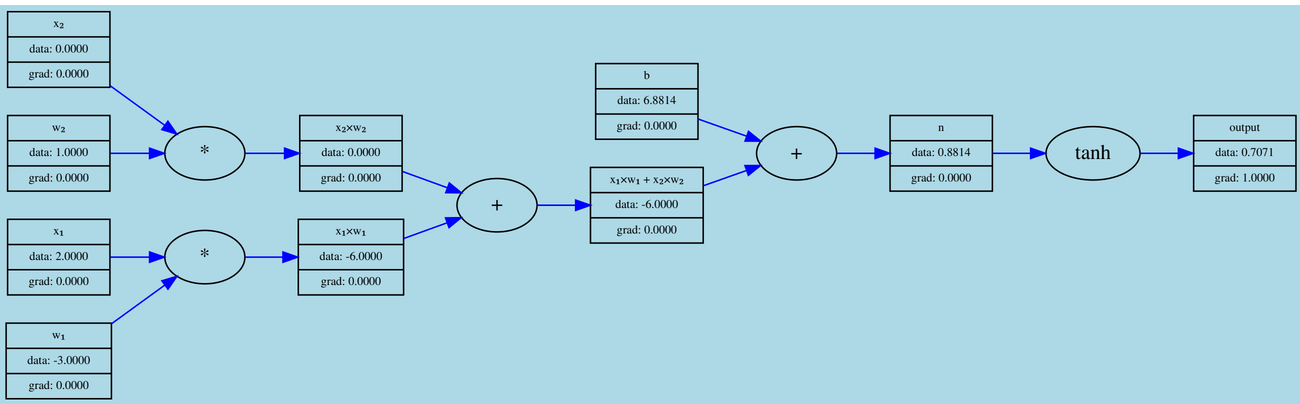
```julia
function try_grad()
    h = 0.001

    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
```

```julia
# got 7 var =>
try_grad()
```

```
one_neuron (generic function with 1 method)
```

```julia
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
```




```julia
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)
```

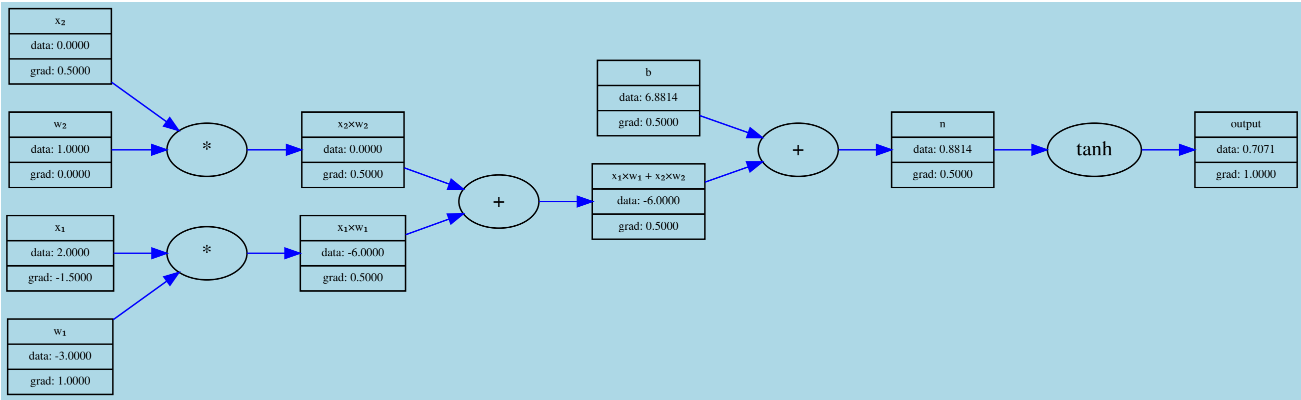
```
begin
    n.grad = 1 - o.data^2

    # we also can fill in the gradient for x₁w₁x₂w₂, b - applying + rule
    x₁w₁x₂w₂.grad = bias.grad = n.grad

    # and for x₁w₁, x₂w₂ - applying + rule
    x₁w₁.grad, x₂w₂.grad = x₁w₁x₂w₂.grad, x₁w₁x₂w₂.grad
end
```

```
(0.5, 0.0)
```

```
x₂.grad, w₂.grad = w₂.data * x₂w₂.grad, x₂.data * x₂w₂.grad
```

```
(-1.5, 1.0)
```

```
x₁.grad, w₁.grad = w₁.data * x₁w₁.grad, x₁.data * x₁w₁.grad
```



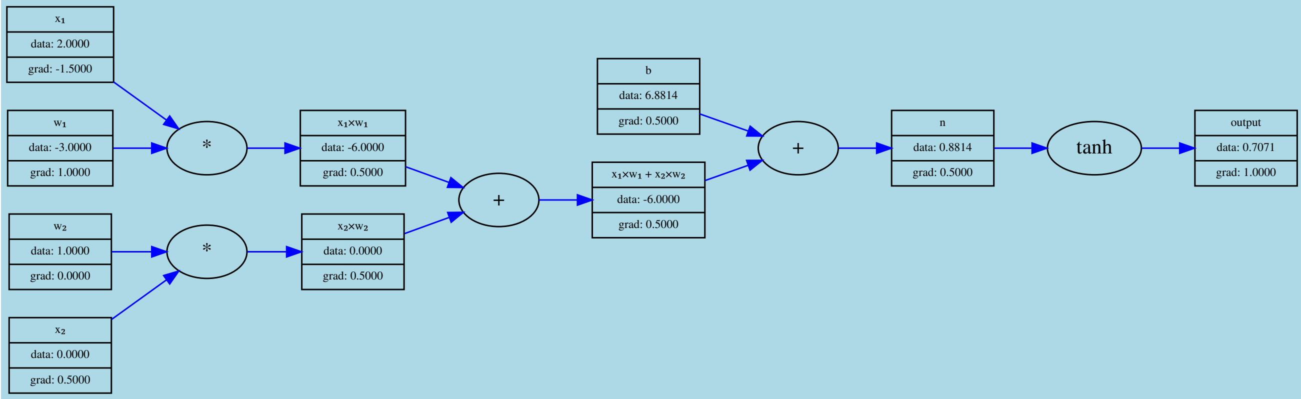

```
# redraw graph with gradient updated
draw_dot(o)
```

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
    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)

    # and now the backward pass
    o.grad = 1.0
    o._backward()
    n._backward()
    x₁w₁x₂w₂._backward()
    x₁w₁._backward()
    x₂w₂._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)
```

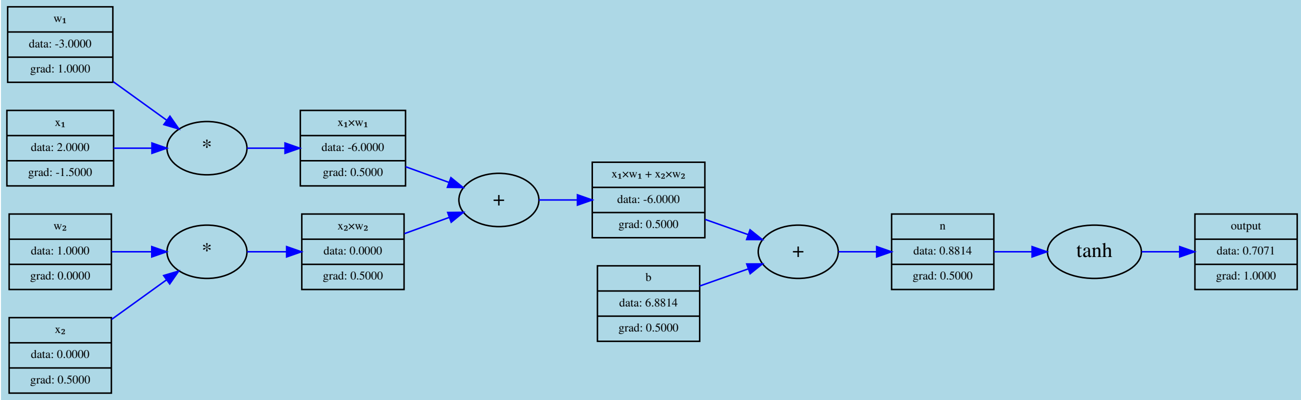
```
function topological_order(o::Value)
    topo, visited = [], Set()
    function build_topological_order(v::Value)
        if v ∉ visited
            push!(visited, v)
            for child ∈ v._prev
                build_topological_order(child)
            end
            push!(topo, v)
       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

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

```
topological_order(o₁)
```

```
auto_backprop_one_neuron (generic function with 1 method)
```

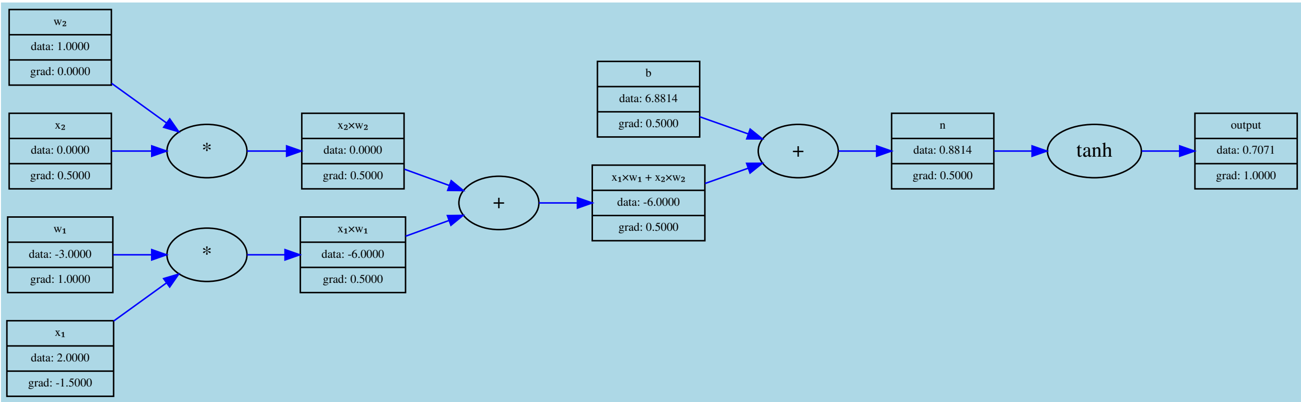
```
function auto_backprop_one_neuron()
    # 2 inputs
    x₁, x₂ = YaValue(2.0; label="x₁"), YaValue(0.0; label="x₂")
    # 2 weights
    w₁, w₂ = YaValue(-3.0; label="w₁"), YaValue(1.0; label="w₂")
    # bias
    b = YaValue(6.8813735870195432; label="b")

    # 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
```



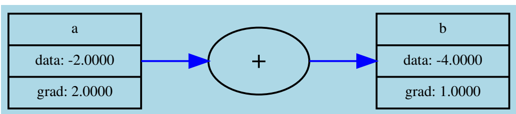

```julia
begin
    o₂ = auto_backprop_one_neuron()
    draw_dot(o₂)
end
```

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

Let's do this...




```julia
begin
    (o₃, _rest_) = one_neuron()
    backward(o₃)
    draw_dot(o₃)
end
```




```julia
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

```julia
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)
```

```julia
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})
```

```julia
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)))
```

```julia
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))
```

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

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

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

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

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

```julia
##
## Extending operator for DataType Value{T
##
for op ∈ (:+, :*)
    @eval begin
        ## Allowing:
        #   - Value{T} :op T  =>  Value{T}
        #   - T :op Value{T}  =>  Value{T}
        ($op)(self::Value{T}, other::T) where {T <: Real} = ($op)(self, Value{T}(other))
        ($op)(other::T, self::Value{T}) where {T <: Real} = ($op)(self, Value{T}(other))

        # Allowing Value{T} :op S =>  Value{T} where S <: T
        ($op)(self::Value{T}, other::S) where {T <: Real, S <: Integer} =
            ($op)(self, Value{T}(T(other)))
        ($op)(other::S, self::Value{T}) where {T <: Real, S <: Integer} =
            ($op)(self, Value{T}(T(other)))

        # Allowing Value{T} :op Value{S} =>  Value{T} where S <: T
        ($op)(self::Value{T}, other::Value{S}) where {T <: Real, S <: Real} =
            ($op)(self, Value{T}(T(other.data)))
        ($op)(other::Value{S}, self::Value{T}) where {T <: Real, S <: Real} =
            ($op)(self, Value{T}(T(other.data)))
    end
end
```

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

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

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

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

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

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

```
(Value(data=4), Value{Float64}, Value{Int32})
```

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

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

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

```
(AbstractFloat, AbstractFloat)
```

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

```
subtypetree (generic function with 3 methods)
```

```julia
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
```

```julia
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
```

```julia
# subtypetree(Integer)
```

```julia
# subtypetree(AbstractFloat)
```

```
Value(data=4.0)
```

```julia
begin
    z₂ = YaValue(2.0)
    2 * z₂
end
```

tanh in terms of exp.

```julia
function Base.exp(self::Value{T}) where {T <: Real}
    x = self.data
    y = YaValue(exp(x); _children=(self, ), _op=:exp, label="exp")
    function _backward_fn()
        self.grad += y.data * y.grad  # because ∂exp/∂x = exp
    end
    y._backward = _backward_fn
    y
end
```

More operators

Note that a / b == a × 1/b == a × b^{-1}

```julia
Base.:/(self::Value{T}, other::Value{T}) where {T <: Real} = Base.:*(self, other^(-1.))
```

```julia
function Base.:^(self::Value{T}, p::T) where {T <: Real}
    y = YaValue(self.data^p; _children=(self, ), _op=:^, label="^p")
    function _backward_fn()
        self.grad += p * self.data^(p - 1) * y.grad # because ∂x^p/∂x = p x^(p -1 )
    end
    y._backward = _backward_fn
    y
end
```

```julia
Base.:-(self::Value{T}, other::Value{T}) where {T <: Real} = Base.:+(self, other * -1.)
```

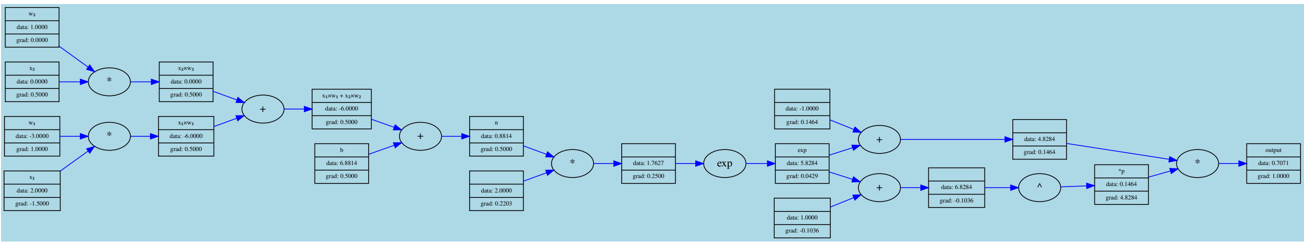
```julia
Base.:-(self::Value{T}, other::S)  where {T <: Real, S <: Real} = Base.:+(self, other * -1.)
```

```
(Value(data=0.5), Value(data=-3.0), Value(data=3.0))
```

```julia
z₂ / YaValue(4.0), z₂ - YaValue(5.0), YaValue(5.0) - 2
```

```
one_neuron_alt (generic function with 1 method)
```

```julia
function one_neuron_alt()
    # 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)
    e = exp(2 * n)
    o = (e - 1) / (e + 1)
    o.label = "output"
    # -----------------------------
    # (o, n, x₁w₁x₂w₂, b, x₁w₁, x₂w₂, x₁, x₂, w₁, w₂)
    o
end
```




```julia
begin
    o₄ = one_neuron_alt()
    backward(o₄)
    draw_dot(o₄)
end
```

Implementing a MLP

Neuron Datatype

```julia
using Random    , Distributions
```

```
TaskLocalRNG()
```

```julia
Random.seed!(42)
```

```julia
# const uniform_d = Uniform(-1, 1) # = Distributions.Uniform{Float64}(a=-1.0, b=1.0)
```

```julia
# const uniform_df32 = Uniform{Float32}(-1, 1) # = Distributions.Uniform{Float32}(a=-1.0f0, b=1.0f0)
```

```julia
# const NT = rand(uniform_d, 1) |> eltype # = Float64
```

```
struct Neuron{T <: AbstractFloat}
    w::Vector{Value{T}}
    b::Value{T}

    function Neuron{T}(n_inp::Integer; dist=Uniform{T}(-1., 1.)) where {T <: AbstractFloat}
        @assert n_inp ≥ 1
        w = Value{T}.(rand(dist, n_inp))
        b = Value{T}(rand(dist, 1)[1])
        new(w, b)
    end

    function Neuron{Float32}(n_inp::Integer; dist=Uniform{Float32}(-1., 1.))
        @assert n_inp ≥ 1
        w = [Value{Float32}(Float32(rand(dist, n_inp)[1])) for _ ∈ 1:n_inp]
        b = Value{Float32}(Float32(rand(dist, 1)[1]))
        new(w, b)
    end
end
```

Neuron_f64 (generic function with 1 method)

```
Neuron_f64(n_inp::Integer) = Neuron{Float64}(n_inp)
```

```
#
## not working as rand will return float64
#
# function Neuron_f32(n_inp::Integer)
#   DT = Float32
#   Neuron{DT}(n_inp; dist=Uniform{DT}(-1., 1.))
# end
```

```
n₁ =
 Neuron([Value(data=0.32971615585069847), Value(data=0.6861712678966163), Value(data=-0.3645970744141882), Value(data=0.337
```

```
n₁ = Neuron_f64(4)
```

```
n₂ =
 Neuron([Value(data=-0.27047643), Value(data=0.83268), Value(data=-0.07984376), Value(data=-0.31266853)], Value(data=0.1743
```

```
n₂ = Neuron{Float32}(4)
```

(Float32, Float64)

```
n₂.w[1].data |> typeof, n₁.w[1].data |> typeof
```

Union{Vector{T}, Array{Value{T}, 1}} where T<:AbstractFloat

```
UVT{T} = Union{Vector{T}, Vector{Value{T}}} where {T <: AbstractFloat}
```

forward

```
forward(...)
eval neuron by taking the dot-product between input and weights, sum, add bias and pass it to activation funct
ion
```

```
"""
    forward(...)
    eval neuron by taking the dot-product between input and weights, sum, add bias and pass it to activation function
"""
function forward(self::Neuron{T}, x::UVT{T}; act_fn=tanh) where {T <: AbstractFloat}
    # x == vector of inputs
    @assert length(self.w) >= 1 && length(self.w) == length(x)
    self.w .* x |> d -> sum(d; init=self.b) |> act_fn
end
```

Value(data=0.989725658082787)

```
forward(n₁, [1., 2., 3., 4.])
```

Layer Datatype

```
struct Layer{T <: AbstractFloat}
    neurons::Vector{Neuron{T}}

    function Layer{T}(n_inp::Integer, n_out::Integer) where {T <: AbstractFloat}method matching forward(::Main.var"wor
        @assert n_inp ≥ 1 && n_out ≥ 1
        vn = [Neuron{T}(n_inp) for _ ∈ 1:n_out]
        new(vn)
    end
end
```

forward (generic function with 1 method)

```
function forward(self::Layer{T}, x::UVT{T}) where {T <: AbstractFloat}
    y = [forward(n, x) for n ∈ self.neurons]
    length(y) == 1 ? y[1] : y
end
```

```
[Value(data=-0.21515723855079233), Value(data=0.9037032021926392), Value(data=0.9427021591283836)]
```

```julia
begin
    xx₁ = [1.1, 2.0]
    nl₁ = Layer{Float64}(2, 3)  # 2 inputs, 3 outputs
    forward(nl₁, xx₁)
end
```

MLP Datatype

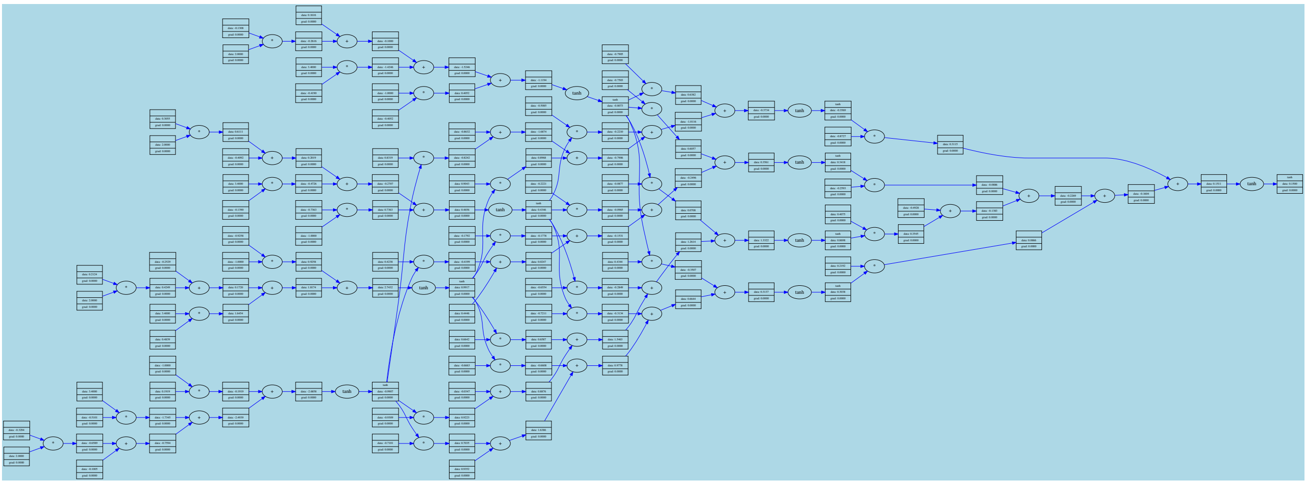
```julia
struct MLP{T <: AbstractFloat}
    layers::Vector{Layer{T}}

    function MLP{T}(n_inp::Integer, n_outs::Vector{<: Integer}) where {T <: AbstractFloat}
        @assert n_inp ≥ 1 && length(n_outs) ≥ 1
        sz = [n_inp, n_outs...]
        layers = [Layer{T}(sz[ix], sz[ix + 1]) for ix ∈ 1:length(n_outs)]
        new(layers)
    end
end
```

```
forward (generic function with 3 methods)
```

```julia
function forward(self::MLP{T}, x::UVT{T}) where {T <: AbstractFloat}
    for layer ∈ self.layers
        x = forward(layer, x) # mutate x
    end
    x
end
```

```
Value(data=0.14997258564896673)
```

```julia
begin
    mlp_x = [2.0, 3.4, -1.0]           # inputs
    mlp = MLP{Float64}(3, [4, 4, 1])  # input 3 neurons, 2 hidden with 4 neurons each, 1 output neuron
    oₓ = forward(mlp, mlp_x)
end
```



```julia
draw_dot(oₓ)
```

```julia
# TBD examples, loss, backprop
```