Zygote 자이곧

May 9, 2019

Contents

Contents i

I Home 1

1 Zygote 3
1.1 Setup . . . 3
1.2 Taking Gradients . . . 3
1.3 Structs and Types . . . 4
1.4 Gradients of ML models . . . 5

II Custom Adjoints 7

2 Custom Adjoints 9
2.1 Pullbacks . . . 9
2.2 Custom Adjoints . . . 10
2.3 Custom Types . . . 10
2.4 Advanced Adjoints . . . 11
Gradient Hooks . . . 11
Checkpointing . . . 12
Gradient Reflection . . . 12

III Utilities 13

3 Utilities 15

IV Complex Differentiation 19

4 Complex Differentiation 21

V Flux 23

5 Flux 25

VI Profiling 27

6 Debugging in Time and Space 29
6.1 Performance Profiling . . . 29
6.2 Memory Profiling . . . 29
6.3 Reflection . . . 30

VII Internals 31

7 Internals 33
7.1 What Zygote Does . . . 33
7.2 Closures . . . 34
7.3 Entry Points . . . 34
7.4 Closure Conversion . . . 35
7.5 Debugging . . . 36

VIII Glossary 37

8 Glossary 39

Part I

Home

Chapter 1

Zygote

Welcome! Zygote extends the Julia language to support differentiable programming. With Zygote you can write down any Julia code you feel like – including using existing Julia packages – then get gradients and optimise your program. Deep learning, ML and probabilistic programming are all different kinds of differentiable programming that you can do with Zygote.

At least, that's the idea. We're still in beta so expect some adventures.

1.1 Setup

Zygote is still moving quickly and it's best to work from the development branches. Run this in a Julia session:

```
using Pkg; pkg"add Zygote#master"
```

1.2 Taking Gradients

Zygote is easy to understand since, at its core, it has a one-function API (`forward`), along with a few simple conveniences. Before explaining `forward`, we'll look at the higher-level function `gradient`.

`gradient` calculates derivatives. For example, the derivative of $3x^2 + 2x + 1$ is $6x + 2$, so when `x = 5`, `dx = 32`.

```
julia> using Zygote

julia> gradient(x -> 3x^2 + 2x + 1, 5)
(32,)
```

`gradient` returns a tuple, with a gradient for each argument to the function.

```
julia> gradient((a, b) -> a*b, 2, 3)
(3, 2)
```

This will work equally well if the arguments are arrays, structs, or any other Julia type, but the function should return a scalar (like a loss or objective l, if you're doing optimisation / ML).

```
julia> W = rand(2, 3); x = rand(3);

julia> gradient(W -> sum(W*x), W)[1]
2×3 Array{Float64,2}:
 0.0462002  0.817608  0.979036
 0.0462002  0.817608  0.979036

julia> gradient(x -> 3x^2 + 2x + 1, 1//4)
(7//2,)
```

Control flow is fully supported, including recursion.

```
julia> function pow(x, n)
         r = 1
         for i = 1:n
           r *= x
         end
         return r
       end
pow (generic function with 1 method)

julia> gradient(x -> pow(x, 3), 5)
(75,)

julia> pow2(x, n) = n <= 0 ? 1 : x*pow(x, n-1)
pow2 (generic function with 1 method)

julia> gradient(x -> pow2(x, 3), 5)
(75,)
```

Data structures are also supported, including mutable ones like dictionaries. Arrays are currently immutable, though this may change in future.

```
julia> d = Dict()
Dict{Any,Any} with 0 entries

julia> gradient(5) do x
         d[:x] = x
         d[:x] * d[:x]
       end
(10,)

julia> d[:x]
5
```

1.3 Structs and Types

Julia makes it easy to work with custom types, and Zygote makes it easy to differentiate them. For example, given a simple `Point` type:

```
import Base: +, -

struct Point
  x::Float64
  y::Float64
end

a::Point + b::Point = Point(a.x + b.x, a.y + b.y)
a::Point - b::Point = Point(a.x - b.x, a.y - b.y)
dist(p::Point) = sqrt(p.x^2 + p.y^2)
```

```
julia> a = Point(1, 2)
Point(1.0, 2.0)

julia> b = Point(3, 4)
Point(3.0, 4.0)

julia> dist(a + b)
7.211102550927978

julia> gradient(a -> dist(a + b), a)[1]
(x = 0.5547001962252291, y = 0.8320502943378437)
```

Zygote's default representation of the "point adjoint" is a named tuple with gradients for both fields, but this can of course be customised too.

This means we can do something very powerful: differentiating through Julia libraries, even if they weren't designed for this. For example, `colordiff` might be a smarter loss function on colours than simple mean-squared-error:

```
julia> using Colors

julia> colordiff(RGB(1, 0, 0), RGB(0, 1, 0))
86.60823557376344

julia> gradient(colordiff, RGB(1, 0, 0), RGB(0, 1, 0))
((r = 0.4590887719632896, g = -9.598786801605689, b = 14.181383399012862), (r = -1.7697549557037275, g =
↪  28.88472330558805, b = -0.044793892637761346))
```

1.4 Gradients of ML models

It's easy to work with even very large and complex models, and there are few ways to do this. Autograd-style models pass around a collection of weights.

```
julia> linear(θ, x) = θ[:W] * x .+ θ[:b]
linear (generic function with 1 method)

julia> x = rand(5);

julia> θ = Dict(:W => rand(2, 5), :b => rand(2))
Dict{Any,Any} with 2 entries:
  :b => [0.0430585, 0.530201]
  :W => [0.923268 … 0.589691]

# Alternatively, use a named tuple or struct rather than a dict.
# θ = (W = rand(2, 5), b = rand(2))

julia> θ = gradient(θ -> sum(linear(θ, x)), θ)[1]
Dict{Any,Any} with 2 entries:
  :b => [1.0, 1.0]
  :W => [0.628998 … 0.433006]
```

An extension of this is the Flux-style model in which we use call overloading to combine the weight object with the forward pass (equivalent to a closure).

```
julia> struct Linear
         W
         b
       end

julia> (l::Linear)(x) = l.W * x .+ l.b

julia> model = Linear(rand(2, 5), rand(2))
Linear([0.267663 … 0.334385], [0.0386873, 0.0203294])

julia> dmodel = gradient(model -> sum(model(x)), model)[1]
(W = [0.652543 … 0.683588], b = [1.0, 1.0])
```

Zygote also support one more way to take gradients, via *implicit parameters* – this is a lot like autograd-style gradients, except we don't have to thread the parameter collection through all our code.

```
julia> W = rand(2, 5); b = rand(2);

julia> linear(x) = W * x .+ b
```

```
linear (generic function with 2 methods)

julia> grads = gradient(() -> sum(linear(x)), Params([W, b]))
Grads(...)

julia> grads[W], grads[b]
([0.652543 … 0.683588], [1.0, 1.0])
```

However, implicit parameters exist mainly for compatibility with Flux's current AD; it's recommended to use the other approaches unless you need this.

Part II

Custom Adjoints

Chapter 2

Custom Adjoints

The `@adjoint` macro is an important part of Zygote's interface; customising your backwards pass is not only possible but widely used and encouraged. While there are specific utilities available for common things like gradient clipping, understanding adjoints will give you the most flexibility. We first give a bit more background on what these pullback things are.

2.1 Pullbacks

`gradient` is really just syntactic sugar around the more fundamental function `forward`.

```
julia> y, back = Zygote.forward(sin, 0.5);

julia> y
0.479425538604203
```

`forward` gives two outputs: the result of the original function, `sin(0.5)`, and a *pullback*, here called `back`. `back` implements the gradient computation for `sin`, accepting a derivative and producing a new one. In mathematical terms, it implements a vector-Jacobian product. Where $y = f(x)$ and the gradient $\frac{\partial l}{\partial x}$ is written $\bar{x}$, the pullback $\mathcal{B}_y$ computes:

$$\bar{x} = \frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} = \mathcal{B}_y(\bar{y})$$

To make this concrete, take the function $y = \sin(x)$. $\frac{\partial y}{\partial x} = \cos(x)$, so the pullback is $\bar{y}\cos(x)$. In other words `forward(sin, x)` behaves the same as

```
dsin(x) = sin(x), y -> (y * cos(x),)
```

`gradient` takes a function $l = f(x)$ and assumes $l = \frac{\partial l}{\partial l} = 1$ and feeds this in to the pullback. In the case of `sin`,

```
julia> function gradsin(x)
         _, back = dsin(x)
         back(1)
       end
gradsin (generic function with 1 method)

julia> gradsin(0.5)
(0.8775825618903728,)

julia> cos(0.5)
0.8775825618903728
```

More generally

```
julia> function mygradient(f, x...)
         _, back = Zygote.forward(f, x...)
         back(1)
       end
mygradient (generic function with 1 method)

julia> mygradient(sin, 0.5)
(0.8775825618903728,)
```

The rest of this section contains more technical detail. It can be skipped if you only need an intuition for pullbacks; you generally won't need to worry about it as a user.

If x and y are vectors, $\frac{\partial y}{\partial x}$ becomes a Jacobian. Importantly, because we are implementing reverse mode we actually left-multiply the Jacobian, i.e. `v' J`, rather than the more usual `J*v`. Transposing `v` to a row vector and back `(v' J)'` is equivalent to `J' v` so our gradient rules actually implement the *adjoint* of the Jacobian. This is relevant even for scalar code: the adjoint for `y = sin(x)` is `x = sin(x)'*y`; the conjugation is usually moot but gives the correct behaviour for complex code. "Pullbacks" are therefore sometimes called "vector-Jacobian products" (VJPs), and we refer to the reverse mode rules themselves as "adjoints".

Zygote has many adjoints for non-mathematical operations such as for indexing and data structures. Though these can still be seen as linear functions of vectors, it's not particularly enlightening to implement them with an actual matrix multiply. In these cases it's easiest to think of the adjoint as a kind of inverse. For example, the gradient of a function that takes a tuple to a struct (e.g. `y = Complex(a, b)`) will generally take a struct to a tuple (`(y.re, y.im)`). The gradient of a `getindex` `y = x[i...]` is a `setindex!` `x[i...] = y`, etc.

2.2 Custom Adjoints

We can extend Zygote to a new function with the `@adjoint` function.

```
julia> mul(a, b) = a*b

julia> using Zygote: @adjoint

julia> @adjoint mul(a, b) = mul(a, b), c -> (c*b, c*a)

julia> gradient(mul, 2, 3)
(3, 2)
```

It might look strange that we write `mul(a, b)` twice here. In this case we want to call the normal `mul` function for the forward pass, but you may also want to modify the forward pass (for example, to capture intermediate results in the pullback).

2.3 Custom Types

One good use for custom adjoints is to customise how your own types behave during differentiation. For example, in our `Point` example we noticed that the adjoint is a named tuple, rather than another point.

```
import Base: +, -

struct Point
  x::Float64
  y::Float64
end

width(p::Point) = p.x
height(p::Point) = p.y

a::Point + b::Point = Point(width(a) + width(b), height(a) + height(b))
a::Point - b::Point = Point(width(a) - width(b), height(a) - height(b))
dist(p::Point) = sqrt(width(p)^2 + height(p)^2)
```

```
julia> gradient(a -> dist(a), Point(1, 2))[1]
(x = 0.5547001962252291, y = 0.8320502943378437)
```

Fundamentally, this happens because of Zygote's default adjoint for `getfield`.

```
julia> gradient(a -> a.x, Point(1, 2))
((x = 1, y = nothing),)
```

We can overload this by modifying the getters `height` and `width`.

```
julia> @adjoint width(p::Point) = p.x, x -> (Point(x, 0),)

julia> @adjoint height(p::Point) = p.y, y -> (Point(0, y),)

julia> Zygote.refresh() # currently needed when defining new adjoints

julia> gradient(a -> height(a), Point(1, 2))
(Point(0.0, 1.0),)

julia> gradient(a -> dist(a), Point(1, 2))[1]
Point(0.4472135954999579, 0.8944271909999159)
```

If you do this you should also overload the `Point` constructor, so that it can handle a `Point` gradient (otherwise this function will error).

```
julia> @adjoint Point(a, b) = Point(a, b), p -> (p.x, p.y)

julia> gradient(x -> dist(Point(x, 1)), 1)
(0.7071067811865475,)
```

2.4 Advanced Adjoints

We usually use custom adjoints to add gradients that Zygote can't derive itself (for example, because they `ccall` to BLAS). But there are some more advanced and fun things we can to with `@adjoint`.

Gradient Hooks

```
julia> hook(f, x) = x
hook (generic function with 1 method)

julia> @adjoint hook(f, x) = x, x -> (nothing, f(x))
```

hook doesn't seem that interesting, as it doesn't do anything. But the fun part is in the adjoint; it's allowing us to apply a function `f` to the gradient of `x`.

```
julia> gradient((a, b) -> hook(-, a)*b, 2, 3)
(-3, 2)
```

We could use this for debugging or modifying gradients (e.g. gradient clipping).

```
julia> gradient((a, b) -> hook(a -> @show(a), a)*b, 2, 3)
ā = 3
(3, 2)
```

Zygote provides both hook and `@showgrad` so you don't have to write these yourself.

Checkpointing

A more advanced example is checkpointing, in which we save memory by re-computing the forward pass of a function during the backwards pass. To wit:

```
julia> checkpoint(f, x) = f(x)
checkpoint (generic function with 1 method)

julia> @adjoint checkpoint(f, x) = f(x), y -> Zygote._forward(f, x)[2](y)

julia> gradient(x -> checkpoint(sin, x), 1)
(0.5403023058681398,)
```

If a function has side effects we'll see that the forward pass happens twice, as expected.

```
julia> foo(x) = (println(x); sin(x))
foo (generic function with 1 method)

julia> gradient(x -> checkpoint(foo, x), 1)
1
1
(0.5403023058681398,)
```

Gradient Reflection

It's easy to check whether the code we're running is currently being differentiated.

```
isderiving() = false

@adjoint isderiving() = true, _ -> nothing
```

A more interesting example is to actually detect how many levels of nesting are going on.

```
nestlevel() = 0

@adjoint nestlevel() = nestlevel()+1, _ -> nothing
```

Demo:

```
julia> function f(x)
         println(nestlevel(), " levels of nesting")
         return x
       end
f (generic function with 1 method)

julia> grad(f, x) = gradient(f, x)[1]
grad (generic function with 1 method)

julia> f(1);
0 levels of nesting

julia> grad(f, 1);
1 levels of nesting

julia> grad(x -> x*grad(f, x), 1);
2 levels of nesting
```

Part III

Utilities

Chapter 3

Utilities

Zygote provides a set of helpful utilities. These are all "user-level" tools – in other words you could have written them easily yourself, but they live in Zygote for convenience.

`Zygote.@showgrad` – Macro.

```
@showgrad(x) -> x
```

Much like `@show`, but shows the gradient about to accumulate to `x`. Useful for debugging gradients.

```
julia> gradient(2, 3) do a, b
         @showgrad(a)*b
       endð
(a) = 3
(3, 2)
```

Note that the gradient depends on how the output of `@showgrad` is *used*, and is not the *overall* gradient of the variable `a`. For example:

```
julia> gradient(2) do a
     @showgrad(a)*a
   endð
(a) = 2
(4,)

julia> gradient(2, 3) do a, b
         @showgrad(a) # not used, so no gradient
         a*b
       endð
(a) = nothing
(3, 2)
```

source

`Zygote.hook` – Function.

```
hook(x -> ..., x) -> x
```

Gradient hooks. Allows you to apply an arbitrary function to the gradient for `x`.

```
julia> gradient(2, 3) do a, b
         hook(a -> @show(a), a)*b
       endā
 = 3
(3, 2)

julia> gradient(2, 3) do a, b
         hook(-, a)*b
       end
(-3, 2)
```

source

Zygote.dropgrad – Function.

```
dropgrad(x) -> x
```

Drop the gradient of x.

```
julia> gradient(2, 3) do a, b
     dropgrad(a)*b
   end
(nothing, 2)
```

source

Zygote.hessian – Function.

```
hessian(f, x)
```

Construct the Hessian of f, where x is a real or real array and f(x) is a real.

```
julia> hessian(((a, b),) -> a*b, [2, 3])
2×2 Array{Int64,2}:
 0  1
 1  0
```

source

Zygote.Buffer – Type.

```
Buffer(xs, ...)
```

Buffer is an array-like type which is mutable when taking gradients. You can construct a Buffer with the same syntax as similar (e.g. Buffer(xs, 5)) and then use normal indexing. Finally, use copy to get back a normal array.

For example:

```
julia> function vstack(xs)
           buf = Buffer(xs, length(xs), 5)
           for i = 1:5
             buf[:, i] = xs
           end
           return copy(buf)
         end
vstack (generic function with 1 method)

julia> vstack([1, 2, 3])
3×5 Array{Int64,2}:
 1  1  1  1  1
 2  2  2  2  2
 3  3  3  3  3

julia> gradient(x -> sum(vstack(x)), [1, 2, 3])
([5.0, 5.0, 5.0],)
```

Buffer is not an AbstractArray and can't be used for linear algebra operations like matrix multiplication. This prevents it from being captured by pullbacks.

copy is a semantic copy, but does not allocate memory. Instead the Buffer is made immutable after copying.

source

Zygote.forwarddiff – Function.

```
forwarddiff(f, x) -> f(x)
```

Runs `f(x)` as usual, but instructs Zygote to differentiate `f` using forward mode, rather than the usual reverse mode.

Forward mode takes time linear in `length(x)` but only has constant memory overhead, and is very efficient for scalars, so in some cases this can be a useful optimisation.

```
julia> function pow(x, n)
         r = one(x)
         for i = 1:n
           r *= x
         end
         return r
       end
pow (generic function with 1 method)

julia> gradient(5) do x
         forwarddiff(x) do x
           pow(x, 2)
         end
       end
(10,)
```

Note that the function `f` will *drop gradients* for any closed-over values.

```
julia> gradient(2, 3) do a, b
         forwarddiff(a) do a
           a*b
         end
       end
(3, nothing)
```

This can be rewritten by explicitly passing through `b`, i.e.

```
gradient(2, 3) do a, b
  forwarddiff([a, b]) do (a, b)
    a*b
  end
end
```

source

Part IV

Complex Differentiation

Chapter 4

Complex Differentiation

Complex numbers add some difficulty to the idea of a "gradient". To talk about `gradient(f, x)` here we need to talk a bit more about `f`.

If `f` returns a real number, things are fairly straightforward. For $c = x+yi$ and $z = f(c)$, we can define the adjoint $\bar{c} = \frac{\partial z}{\partial x} + \frac{\partial z}{\partial y} i = \bar{x}+\bar{y}i$ (note that $\bar{c}$ means gradient, and c' means conjugate). It's exactly as if the complex number were just a pair of reals `(re, im)`. This works out of the box.

```
julia> gradient(c -> abs2(c), 1+2im)
(2 + 4im,)
```

However, while this is a very pragmatic definition that works great for gradient descent, it's not quite aligned with the mathematical notion of the derivative: i.e. $f(c + \epsilon) \approx f(c) + \bar{c}\epsilon$. In general, such a $\bar{c}$ is not possible for complex numbers except when `f` is *holomorphic* (or *analytic*). Roughly speaking this means that the function is defined over `c` as if it were a normal real number, without exploiting its complex structure – it can't use `real`, `imag`, `conj`, or anything that depends on these like `abs2` (`abs2(x) = x*x'`). (This constraint also means there's no overlap with the Real case above; holomorphic functions always return complex numbers for complex input.) But most "normal" numerical functions – `exp`, `log`, anything that can be represented by a Taylor series – are fine.

Fortunately it's also possible to get these derivatives; they are the conjugate of the gradients for the real part.

```
julia> gradient(x -> real(log(x)), 1+2im)[1] |> conj
0.2 - 0.4im
```

We can check that this function is holomorphic – and thus that the gradient we got out is sensible – by checking the Cauchy-Riemann equations. In other words this should give the same answer:

```
julia> -im*gradient(x -> imag(log(x)), 1+2im)[1] |> conj
0.2 - 0.4im
```

Notice that this fails in a non-holomorphic case, `f(x) = log(x')`:

```
julia> gradient(x -> real(log(x')), 1+2im)[1] |> conj
0.2 - 0.4im

julia> -im*gradient(x -> imag(log(x')), 1+2im)[1] |> conj
-0.2 + 0.4im
```

In cases like these, all bets are off. The gradient can only be described with more information; either a 2x2 Jacobian (a generalisation of the Real case, where the second column is now non-zero), or by the two Wirtinger derivatives (a generalisation of the holomorphic case, where $\frac{f}{z'}$ is now non-zero). To get these efficiently, as we would a Jacobian, we can just call the backpropagators twice.

```
function jacobi(f, x)
  y, back = Zygote.forward(f, x)
  back(1)[1], back(im)[1]
end

function wirtinger(f, x)
  du, dv = jacobi(f, x)
  (du' + im*dv')/2, (du + im*dv)/2
end
```

```
julia> wirtinger(x -> 3x^2 + 2x + 1, 1+2im)
(8.0 + 12.0im, 0.0 + 0.0im)

julia> wirtinger(x -> abs2(x), 1+2im)
(1.0 - 2.0im, 1.0 + 2.0im)
```

Part V

Flux

Chapter 5

Flux

It's easy to use Zygote in place of Flux's default AD, Tracker, just by changing `Tracker.gradient` to `Zygote.gradient`. The API is otherwise the same.

```
julia> using Flux, Zygote

julia> m = Chain(Dense(10, 5, relu), Dense(5, 2))
Chain(Dense(10, 5, NNlib.relu), Dense(5, 2))

julia> x = rand(10);

julia> gs = gradient(() -> sum(m(x)), params(m))
Grads(...)

julia> gs[m[1].W]
5×10 Array{Float32,2}:
 -0.255175  -1.2295   ...
```

You can use optimisers and update gradients as usual.

```
julia> opt = ADAM();

julia> Flux.Optimise.update!(opt, params(m), gs)
```

Part VI

Profiling

Chapter 6

Debugging in Time and Space

Because Zygote generates Julia code for the backwards pass, many of Julia's normal profiling and performance debugging tools work well on it out of the box.

6.1 Performance Profiling

Julia's sampling profiler is useful for understanding performance. We recommend running the profiler in Juno, but the terminal or ProfileView.jl also work well.

The bars indicate time taken in both the forwards and backwards passes at that line. The canopy chart on the right shows us each function call as a block, arranged so that when `f` calls `g`, `g` gets a block just below `f`, which is bigger the longer it took to run. If we dig down the call stack we'll eventually find the adjoints for things like `matmul`, which we can click on to view.

The trace inside the adjoint can be used to distinguish time taken by the forwards and backwards passes.

6.2 Memory Profiling

Reverse-mode AD typically uses memory proportional to the number of operations in the program, so long-running programs can also suffer memory usage issues. Zygote includes a space profiler to help debug these issues. Like the time profiler, it shows a canopy chart, but this time hovering over it displays the number of bytes stored by each line of the program.

Note that this currently only works inside Juno.

Figure 6.1:

Figure 6.2:

Figure 6.3:

6.3 Reflection

Julia's code and type inference reflection tools can also be useful, though Zygote's use of closures can make the output noisy. To see the code Julia runs you should use the low-level `_forward` method and the pullback it returns. This will directly show either the derived adjoint code or the code for a custom adjoint, if there is one.

```
julia> using Zygote: Context, _forward

julia> add(a, b) = a+b

julia> @code_typed _forward(Context(), add, 1, 2)
CodeInfo(
1 — %1 = (Base.getfield)(args, 1)::Int64
|   %2 = (Base.getfield)(args, 2)::Int64
|   %3 = (Base.add_int)(%1, %2)::Int64
|   %4 = (Base.tuple)(%3, $(QuoteNode(∂(add))))::PartialTuple(Tuple{Int64,typeof(∂(add))}, Any[Int64,
↪  Const(∂(add), false)])
└──       return %4
) => Tuple{Int64,typeof(∂(add))}

julia> y, back = _forward(Context(), add, 1, 2)
(3, ∂(add))

julia> @code_typed back(1)
CodeInfo(
1 — %1 = (Base.mul_int)(Δ, 1)::Int64
|   %2 = (Base.mul_int)(Δ, 1)::Int64
|   %3 = (Zygote.tuple)(nothing, %1, %2)::PartialTuple(Tuple{Nothing,Int64,Int64}, Any[Const(nothing, false),
↪  Int64, Int64])
└──       return %3
) => Tuple{Nothing,Int64,Int64}
```

Part VII

Internals

Chapter 7

Internals

7.1 What Zygote Does

These notebooks and the Zygote paper provide useful background on Zygote's transform; this page is particularly focused on implementation details.

Given a function like

```
function foo(x)
  a = bar(x)
  b = baz(a)
  return b
end
```

how do we differentiate it? The key is that we can differentiate `foo` if we can differentiate `bar` and `baz`. If we assume we can get pullbacks for those functions, the pullback for `foo` looks as follows.

```
function J(::typeof(foo), x)
  a, da = J(bar, x)
  b, db = J(baz, a)
  return b, function(b)
    a = db(b)
    x = da(a)
    return x
  end
end
```

Thus, where the forward pass calculates `x -> a -> b`, the backwards takes `b -> a -> x` via the pullbacks. The AD transform is recursive; we'll differentiate `bar` and `baz` in the same way, until we hit a case where gradient is explicitly defined.

Here's a working example that illustrates the concepts.

```
J(::typeof(sin), x) = sin(x), y -> y*cos(x)
J(::typeof(cos), x) = cos(x), y -> -y*sin(x)

foo(x) = sin(cos(x))

function J(::typeof(foo), x)
  a, da = J(sin, x)
  b, db = J(cos, a)
  return b, function(b)
    a = db(b)
    x = da(a)
    return x
  end
end
```

```
gradient(f, x) = J(f, x)[2](1)

gradient(foo, 1)
```

Now, clearly this is a mechanical transformation, so the only remaining thing is to automate it – a small matter of programming.

7.2 Closures

The J function here corresponds to `forward` in Zygote. However, `forward` actually a wrapper around the lower level `_forward` function.

```
julia> y, back = Zygote._forward(sin, 0.5);

julia> back(1)
(nothing, 0.8775825618903728)
```

Why the extra `nothing` here? This actually represents the gradient of the function `sin`. This is often `nothing`, but when we have closures the function contains data we need gradients for.

```
julia> f = let a = 3; x -> x*a; end
#19 (generic function with 1 method)

julia> y, back = Zygote._forward(f, 2);

julia> back(1)
((a = 2,), 3)
```

This is a minor point for the most part, but `_forward` will come up in future examples.

7.3 Entry Points

We could do this transform with a macro, but don't want to require that all differentiable code is annotated. Instead a generated function gets us much of the power of a macro without this annotation, because we can use it to get lowered code for a function. We can then modify the code as we please and return it to implement `J(foo, x)`.

```
julia> foo(x) = baz(bar(x))
foo (generic function with 1 method)

julia> @code_lowered foo(1)
CodeInfo(
1 ─ %1 = (Main.bar)(x)
│   %2 = (Main.baz)(%1)
└──      return %2
```

We convert the code to SSA form using Julia's built-in IR data structure, after which it looks like this.

```
julia> Zygote.@code_ir foo(1)
1 1 ─ %1 = (Main.bar)(_2)::Any
  │   %2 = (Main.baz)(%1)::Any
  └──      return %2
```

(There isn't much difference unless there's some control flow.)

The code is then differentiated by the code in `compiler/reverse.jl`. You can see the output with `@code_adjoint`.

```
julia> Zygote.@code_adjoint foo(1)
1 1 ─ %1  = (Zygote._forward)(_2, Zygote.unwrap, Main.bar)::Any
  │   %2  = (Base.getindex)(%1, 1)::Any
  │         (Base.getindex)(%1, 2)::Any
  │   %4  = (Zygote._forward)(_2, %2, _4)::Any
  │   %5  = (Base.getindex)(%4, 1)::Any
  │         (Base.getindex)(%4, 2)::Any
  │   %7  = (Zygote._forward)(_2, Zygote.unwrap, Main.baz)::Any
  │   %8  = (Base.getindex)(%7, 1)::Any
  │         (Base.getindex)(%7, 2)::Any
  │   %10 = (Zygote._forward)(_2, %8, %5)::Any
  │   %11 = (Base.getindex)(%10, 1)::Any
  │         (Base.getindex)(%10, 2)::Any
  └──       return %11
  1 ─ %1  = Δ()::Any
1 │   %2  = (@12)(%1)::Any
  │   %3  = (Zygote.gradindex)(%2, 1)::Any
  │   %4  = (Zygote.gradindex)(%2, 2)::Any
  │         (@9)(%3)::Any
  │   %6  = (@6)(%4)::Any
  │   %7  = (Zygote.gradindex)(%6, 1)::Any
  │   %8  = (Zygote.gradindex)(%6, 2)::Any
  │         (@3)(%7)::Any
  │   %10 = (Zygote.tuple)(nothing, %8)::Any
  └──       return %10
, [1])
```

This code is quite verbose, mainly due to all the tuple unpacking (`gradindex` is just like `getindex`, but handles `nothing` gracefully). The are two pieces of IR here, one for the modified forward pass and one for the pullback closure. The `@` nodes allow the closure to refer to values from the forward pass, and the `Δ()` represents the incoming gradient `y`. In essence, this is just what we wrote above by hand for `J(::typeof(foo), x)`.

`compiler/emit.jl` lowers this code into runnable IR (e.g. by turning `@` references into `getfield`s and stacks), and it's then turned back into lowered code for Julia to run.

7.4 Closure Conversion

There are no closures in lowered Julia code, so we can't actually emit one directly in lowered code. To work around this we have a trick: we have a generic struct like

```
struct Pullback{F}
  data
end
```

We can put whatever we want in `data`, and the `F` will be the signature for the *original* call, like `Tuple{typeof(foo),Int}`. When the pullback gets called it hits another generated function which emits the pullback code.

In hand written code this would look like:

```
struct Pullback{F}
  data
end

function J(::typeof(foo), x)
  a, da = J(sin, x)
  b, db = J(cos, a)
  return b, Pullback{typeof(foo)}((da, db))
end

function(p::Pullback{typeof(foo)})(b)
```

```
  da, db = p.data[1], p.data[2]
  a = db(b)
  x = da(a)
  return x
end
```

7.5 Debugging

Say some of our code is throwing an error.

```
bad(x) = x

Zygote.@adjoint bad(x) = x, _ -> error("bad")

foo(x) = bad(sin(x))

gradient(foo, 1) # error!
```

Zygote can usually give a stacktrace pointing right to the issue here, but in some cases there are compiler crashes that make this harder. In these cases it's best to (a) use `_forward` and (b) take advantage of Zygote's recursion to narrow down the problem function.

```
julia> y, back = Zygote._forward(foo, 1);

julia> back(1) # just make up a value here, it just needs to look similar to `y`
ERROR: bad

# Ok, so we try functions that foo calls

julia> y, back = Zygote._forward(sin, 1);

julia> back(1)
(nothing, 0.5403023058681398)

# Looks like that's fine

julia> y, back = Zygote._forward(bad, 1);

julia> back(1) # ok, here's our issue. Lather, rinse, repeat.
ERROR: bad
```

Of course, our goal is that you never have to do this, but until Zygote is more mature it can be a useful way to narrow down test cases.

Part VIII

Glossary

Chapter 8

Glossary

Differentiation is a minefield of conflicting and overlapping terminology, partly because the ideas have been re-discovered in many different fields (e.g. calculus and differential geometry, the traditional AD community, deep learning, finance, etc.) Many of these terms are not well-defined and others may disagree on the details. Nevertheless, we aim to at least say how *we* use these terms, which will be helpful when reading over Zygote issues, discussions and source code.

The list is certainly not complete; if you see new terms you'd like defined, or would like to add one yourself, please do open an issue or PR.

Adjoint: See *pullback*. Used when defining new pullbacks (i.e. the `@adjoint` macro) since this involves defining the adjoint of the Jacobian, in most cases.

Backpropagation: Essentially equivalent to "reverse-mode AD". Used particularly in the machine learning world to refer to simple chains of functions `f(g(h(x)))`, but has generalised beyond that.

Derivative: Given a scalar function $y = f(x)$, the derivative is $\frac{\partial y}{\partial x}$. "Partial" is taken for granted in AD; there's no interesting distinction between partial and total derivatives for our purposes. It's all in the eye of the beholder.

Differential: Given a function $f(x)$, the linearisation ∂f such that $f(x + \epsilon) \approx f(x) + \partial f \epsilon$. This is a generalisation of the derivative since it applies to, for example, vector-to-vector functions (∂f is a Jacobian) and holomorphic complex functions (∂f is the first Wirtinger derivative). This is *not*, in general, what Zygote calculates, though differentials can usually be derived from gradients.

IR: Intermediate Representation. Essentially source code, but usually lower level – e.g. control flow constructs like loops and branches have all been replaced by `goto`s. The idea is that it's harder for humans to read/write but easier to manipulate programmatically. Worth looking at SSA form as a paradigmatic example.

Gradient: See *sensitivity*. There is no technical difference in Zygote's view, though "gradient" sometimes distinguishes the sensitivity we actually want from e.g. the internal ones that Zygote produces as it backpropagates.

Graph: ML people tend to think of models as "computation graphs", but this is no more true than any program is a graph. In fact, pretty much anything is a graph if you squint hard enough. This also refers to the data structure that e.g. TensorFlow and PyTorch build to represent your model, but see *trace* for that.

Pullback: Given $y = f(x)$ the function $\bar{x} = back(\bar{y})$. In other words, the function `back` in `y, back = Zygote.forward(f, x)`.

Sensitivity: Used to refer to the gradient $\bar{x} = \frac{\partial l}{\partial x}$ with some scalar loss l. In other words, you have a value x (which need not be scalar) at some point in your program, and $\bar{x}$ tells you how you should change that value to decrease the loss. In the AD world, sometimes used to refer to adjoint rules.

Source to Source Differentiation: Or Source Code Transformation (SCT). As opposed to *tracing* programs to simplify them, an alternative is to operate directly on a language's source code or IR, generating new source code for pullbacks. This describes Zygote, Swift for TensorFlow, Tapenade and a few other old ADs that worked on C source files. Zygote and Swift are unusual in that they work on in-memory IR rather than text source.

To an extent, tracing ADs can be viewed as source transform of a Wengert list / trace. The key difference is that the trace is a lossy representation of the original semantics, which causes problems with e.g. control flow. Systems which can preserve some of those semantics (e.g. autograph) begin to blur the line here, though they are still not nearly as expressive as language IRs.

Symbolic Differentiation: Used to refer to differentiation of "mathematical expressions", that is, things like `3x^2 + sin(x)`. Often distinguished from AD, though this is somewhat arbitrary; you can happily produce a symbolic adjoint for a Wengert list, the only difference being that you're allowed to make variable bindings. So it's really just a special case of AD on an unusually limited language.

Tape: This term can refer to pretty much any part of an AD implementation. In particular confusion is caused by conflating the *trace* with the set of values sometimes closed over by a *pullback*. Autograd has a combined trace/closure data structure which is usually described as the tape. On the other hand, PyTorch described their implementation as tape-free because the trace/closure is stored as a DAG rather than a vector, so basically all bets are off here.

Trace: A recording of each mathematical operation used by a program, made at runtime and usually forming a Wengert list. Traces may or may not also record actual runtime values (e.g. PyTorch vs. TensorFlow). They can often be treated as an IR and compiled, but are distinguished from true IRs in that they unroll and inline all control flow, functions and data structures. The tracing process can be thought of as a kind of partial evaluation, though tracers are typically much less worried about losing information.

vector-Jacobian product: see *pullback*. So called because all pullbacks are linear functions that can be represented by (left) multiplication with the Jacobian matrix.

Wengert List: A set of simple variable assignments and mathematical expressions, forming a directed graph. Can be thought of as a limited programming language with variable bindings and numerical functions but no control flow or data structures. If you *trace* a program for AD it will typically take this form.