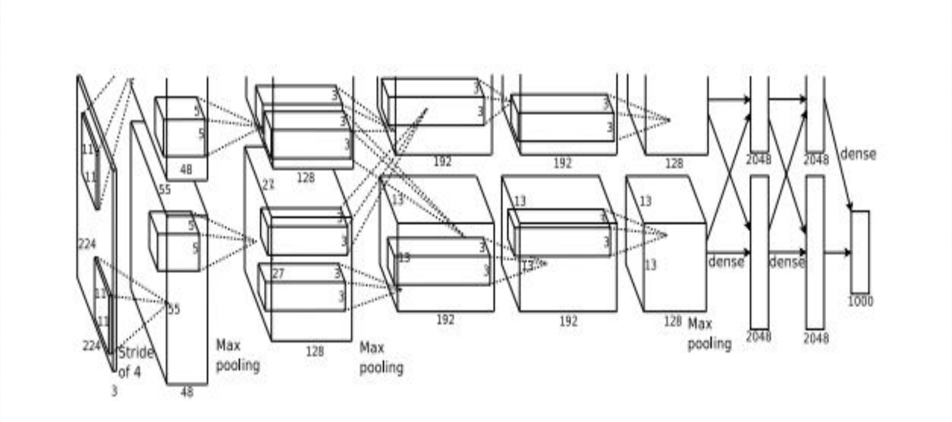
Modular Automatic Differentiation with higher-order functions

Dimitrios Vytiniotis (dvytin@google.com)
Based on work and ideas of many from Google and DeepMind: Richard Wei, Parker Schuh, Marc Rasi, James Bradbury, Dan Zheng, Dougal MacLaurin, Matthew Johnson, Gordon Plotkin, Matthew Willson, Robert Stanforth, DM Performance and ML Programming Teams, and the Swift for Tensorflow project

DeepMind

First few decades of deep learning



Models programmed with text files, configuration scripts and built-in procedures (e.g. stochastic gradient descent variants)

```
name: "AlexNet"
layer {
 name: "data"
 type: "Input"
 top: "data"
 input_param { shape: … }
}
layer {
 name: "conv1"
 ...
 }
layer {
 name: “relu1”
 ...
}
```

The era of differentiable programming

- Custom optimizers and second-order optimization methods (e.g. K-FAC)
- Optimization through traditional algorithms, e.g. parsing and dynamic programming
- Differentiation for custom loss functions, e.g. Conditional Random Fields
- Differentiable interpreters, neural Turing machines
- Data dependent control and data flow, e.g. graph neural networks
- Custom gradient checkpointing, reinforcement learning, …

AD support in TensorFlow, PyTorch, Julia, Jax, DiffSharp, and older systems like Stalingrad, Vlad, Tapenade, and more

Swift for Tensorflow: language support for AD

```
for epoch in 1...epochCount {
   for i in 0 ..< Int(labels.shape[0]) / batchSize {
       let x = minibatch(in: images, at: i)
       let y = minibatch(in: numericLabels, at: i)
       // Compute the gradient with respect to the model.
       let ∇model = classifier.gradient { classifier -> Tensor<Float> in
           let ŷ = classifier(x)
           let loss = softmaxCrossEntropy(logits: ŷ, labels: y)
           return loss }
       optimizer.update(&classifier.allDifferentiableVariables, along: ∇model)
   }
```

Differentiation operator on closure:
`(fun classifier => … return loss)`

www.tensorflow.org/swift
https://github.com/apple/swift/tree/tensorflow

The essence of AD in Swift for Tensorflow (S4TF)

An ahead-of-time (compile-time) symbolic AD phase

∇ NOT an operator on syntax trees!

- Every differentiable function definition: `f(x1:T1,...xn:Tn) : R` of type `(T1,..,Tn) -> R` is *compiled* to a data structure **f** of type `(T1,...Tn) => R`, a "bundle"

`-◦`: linear function
`f(0) = 0`
`f(x1+x2) = f(x1)+f(x2)`

- `T1,...,Tn => R` is just:

Jacobian-Vector product

-
```
T1,...,Tn -> (R,  { derivative : (T1.TangentVector,...,
                                  Tn.TangentVector) -◦ R.TangentVector,
                    pullback : R.TangentVector -◦
                               (T1.TangentVector,...,Tn.TangentVector })
```

Vector-Jacobian product

- Bundle f can be (1) applied, or (2) passed in to other functions, or even (3) *partially applied*

(Co)-Tangent Spaces

`T1,...,Tn => R` is just:

```
T1,...,Tn -> (R,  { derivative : (T1.TangentVector,...,
                                  Tn.TangentVector) -> R.TangentVector,
                    pullback : R.TangentVector ->
                               (T1.TangentVector,...,Tn.TangentVector) } )
```

What is T.TangentVector?

- In S4TF every differentiable type T defines a space of perturbations through an associated type in a Swift Differentiable protocol (a bit like a Haskell type class)
- In math (and in some interpretations of differentiation for higher-order functions) there's also an separate notion of a CoTangentVector, but (like Swift) we will not be making the distinction.
- Just for *brevity* of notation we will use a G[.] type operator to denote the space of perturbations:

```
T1,...,Tn -> (R,  { derivative : (G[T1],..., G[Tn]) -> G[R],
                    pullback : G[R] -> (G[T1],...,G[Tn]) } )
```

(Co)-Tangent Spaces continued

`T1,...,Tn => R` defined as:

```
T1,...,Tn -> (R, { derivative : (G[T1],..., G[Tn]) -> G[R],
                   pullback : G[R] -> (G[T1],...,G[Tn]) } )
```

```
G[Real] = Real
zero = …
sum = …

G[(T1,T2)] = (G[T1], G[T2])
zero = …
sum = …

G[Tensor] = Tensor
zero = …
sum = …
```

For first-order types G[T] = T!

Hence Conal Elliott here dispenses with `G[.]`:

```
D :: (A -> B) -> (A -> (B, A -◦ B))
```

… Is Conal right? …

Recap: Reverse-mode AD in one slide

(we will focus for the rest of the talk on reverse mode AD)

```
// Let f be a function bundle:
f : (Float,Float) => Float

// Let's try to differentiate:
func g(x:Float, y:Float) : Float {
  let (y1,y2) = dup(y);
  let v       = f(x,y1);
  let r       = f(v,y2);
  return r;
}
```

```
func dup(x) { return (x,x); }
func dup(x) { return ((x,x), (g1,g2) in g1+g2) }
```

```
// Recall (just doing reverse-mode AD for simplicity):
f :(Float,Float) -> (Float, G[Float] -> (G[Float],G[Float]))

func g(x:Float, y:Float) {
  let ((y1,y2),pb_dup) = dup(y);
  let (v, pb_f1)       = f(x,y1);
  let (r, pb_f2)       = f(v,y2);
  return (r, gt in {
     let (gv,gy2) = pb_f2(gt)
     let (gx,gy1) = pb_f1(gv)
     let gy       = pb_dup(gy1, gy2)
     return (gx, gy);
     })
}
// Hence we can produce g :: (Float,Float) => Float
```

Note: `x in body` is just Swift notation for `\x -> body`

- Progressively convert every `f(x1..xn)` of a differentiable function `f` to `f(x1..xn)`
- Compose pullbacks in the opposite direction

Enter partial applications

Higher-order functions are an essential part of general purpose programming languages

```
func f(x : Tensor) : Tensor -> Tensor {
  return (y in x*y + x)
}
================================================
⇒ in SIL:
================================================
func clos_1(y x : Tensor) : Tensor {
 return (x*y + x);
func f(x : Tensor) : Tensor -> Tensor {
 return papply(clos_1,x);
}
```

If we have built somehow a bundle for `clos_1` then we want “f” to **return** a bundle for the partial application!

```
struct Model {
   Tensor w;
   func call(x:Tensor):Tensor { return (x*w); }
}
… use site …
mnist.call(inputs);

================================================
⇒ in SIL:
================================================
func call_1(x: Tensor, self : Model) : Tensor {
    return (x * self.w);
}
… use site …
h = papply(call_1,mnist)
r = h(inputs)
```

If we have built somehow a bundle for `call_1` then we want **`papply`**`(call_1,mnist)` to **return** a bundle for the partial application!

```
papply : ((T1..Tn,S1..Sn) -> R, S1..Sn) -> (T1..Tn) -> R
```

Need: a differentiable partial application

```
func f(x : Tensor) : Tensor -> Tensor {
  return (y in x*y + x)
}
================================================
⇒ in SIL:
================================================
// clos_1 : (Tensor,Tensor) => Tensor
func clos_1(y x : Tensor) : Tensor {
  return (x*y + x);
}
// f : Tensor => Tensor => Tensor
func f(x : Tensor) : Tensor => Tensor {
  papply(clos_1,x)
}
```

```
struct Model {
   Tensor w;
   func call(x:Tensor) : Tensor { return (x*w);}
}
… use site …
mnist.call(inputs);

================================================
⇒ in SIL:
================================================
// call_1 : (Tensor,Tensor) => Tensor
func call_1(x: Tensor, self : Model) : Tensor {
    return (x * self.w);
}
… use site …
h = papply(call_1,mnist) : Tensor => Tensor
r = h(inputs)
```

```
papply : ((T1..Tn,S1..Sn) => R, S1..Sn) => (T1..Tn) => R
```

Higher-order arguments equally important

```
func f(x : Tensor, xs : Array Tensor) : Array Tensor {
  let g = y in { x*y + x }
  return Array.map(g, xs)
}
================================================
⇒ in SIL:
================================================
// clos_1 : (Tensor,Tensor) => Tensor
func clos_1(y x : Tensor) : Tensor { return (x*y + x); }
func f(x : Tensor, xs : Array<Tensor>) : Array<Tensor> {
  g = papply(clos_1,x);
  return Array.map(g,xs);
}
// Must have created bundle:
Array.map : (Tensor => Tensor, Array<Tensor>) => Array<Tensor>
```

G[(T1..Tn) => S] = ?

A differentiable papply (aka: curry)

NOTE: switching notation to Haskell, same concepts

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb gsr = ???????
```

- Need to produce a G[T]
- Hence must invoke f's pullback somehow (of type `G[R] => (G[T],G[S])`)
- Hence must "cook up" a `G[R]`
- … but also an actual `S` to get to the pullback -- we only have a `(t : T)` in scope, but nothing of type S!

First attempt:

```
G[S => R] = (S, G[R])
new_pb (s,gr) = fst (snd (f(t,s)) gr)
```

Does this work*?

(*) and BTW what does "work" mean??

A differentiable papply (aka: curry)

A failed attempt

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb gsr = ???????
```

```
pi_left :: (T,S) => T
pi_left (t,s) = (t, \(g : G[T]) -> (g,zero)

fanout :: T => (T, T)
fanout t = ((t,t), \(g1,g2) -> (sum g1 g2)
```

We must be able to define 0 and (+) on G[T], for any T, **including** function types: S=>R.

```
zero :: (S, G[R])
zero = … ??? …    // No way we can define this!

sum :: (S, G[R]) -> (S,G[R]) -> (S, G[R])
sum = … ??? …  // No way we can define this!
```

First attempt:

```
    G[S => R] = (S, G[R])
    new_pb (s,gr) = fst (snd (f(t,s)) gr)
```

A differentiable papply (aka: curry)

Refining the failed attempt

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb gsr = ???????
```

```
pi_left :: ((T,S) => T)
pi_left (t,s) = (t, \(g : G[T]) -> (g,zero)

fanout :: T => (T, T)
fanout t = ((t,t), \(g1,g2) -> (sum g1 g2)
```

```
G[S => R] = List (S, G[R])


new_pb ss_grs =
   List.sum (List.map (\(s,gr) -> fst (snd (f(t,s)) ss_grs)
```

We must be able to define 0 and (+) on G[T], for any T, **including** function types: S=>R.

```
zero :: List (S,G[R])
zero = List.empty // Imposing a monoid structure

sum :: List(S,G[R]) -> List(S,G[R]) -> List(S,G[R])
sum = List.append // Imposing a monoid structure
```

A differentiable papply (aka: curry)

Does this “work”? Category theory to the rescue

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb ss_grs = List.sum $
             List.map (\(s,gr) -> fst (snd (f(t,s)) ss_grs
       in (new_g, new_pb)
```

```
G[S => R] = List (S, G[R])
```

```
eval :: (T => S, T) => S
eval = ...
```

```
(.) :: (T => S) -> (S => R) -> (T => R)
(.) = ...
```

```
id :: (T => T)
id =
```

```
proj_left :: ((T,S) => T)
proj_left = ...
```

```
proj_right :: ((T,S) => S)
proj_right = ...
```

```
tup :: (X => A) -> (Y => B) -> ((X,Y) => (A,B))
tup = …
```

Thm: for `f:(T,S) => R, h : T => S => R`

- `(tuple (curry f) id) . eval ≅ f`
- `curry ((tuple h id)) . eval) ≅ h`

\+ Other usual laws of category theory (i.e. we can form a Cartesian **Closed** Category out of (=>) morphisms

A working solution?

Proof formalized in Coq

Thm: for `f:(T,S) => R, h : T => (S => R)`
- `(tuple (curry f) id) . eval ≅ f`
- `curry ((tuple h id)) . eval) ≅ h`

Proof requires `(≅)` on co-tangent spaces. So when is:

`x ≅ y : G[S=>R]`

It turns out that `G[S=>R] = List(S,G[R])` must behave like an “additive map” e.g:

```
(x,gx1):(x,gx2):xs ≅ (x,gx1 + gx2):xs
(x,zero):xs ≅ xs
```

See formalization for full technical details.

(*) Incidentally for forward-mode AD we need a <u>different</u> G[S=>R] definition. Not going to cover in this talk.

Theorems say that β-laws hold, and η-laws hold.

i.e. if you have a program accepting and returning first-order types, but uses partial applications internally, the program is going to be equivalent (through AD) as if we had fully inlined all intermediate partial applications

Hence this solution “**works**” (*)

… but is it a *workable* solution?

Main appeal: cotangent spaces simple type-level functions of primal types:

```
G[R => S] = List (R, G[S])
```

Main problem: inefficient!

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb ss_grs = List.sum $
             List.map (\(s,gr) -> fst (snd (f(t,s)) ss_grs
       in (new_g, new_pb)
```

We throw half of the returned value of the pullback away!

We end up calling `f` and recomputing its (primal) value, to then just throw it away, many times!

An solution inspired by implicit closure conversion

Any first-class closure `f : T -> S` is really an object `Closure T S`:

```
data Closure T S where
   MkClosure :: Env -> StaticPtr (Env -> T -> S) -> Closure T S
```

Where `Env` is some (existentially quantified) environment and `StaticPtr (Env -> T -> S)` is a mere code pointer -- the entry of a closed function.

Key insight: Make cotangent spaces <u>dependent</u> on the primal value ***itself***, instead of dependent on just the primal value ***type***.

If `(f : T -> S)` was actually a `(Closure env f_static)` then set `G[f : T -> S] = G env`

Why? Because `f_static` is just a constant, it can't vary!

Idea appears in Pearlmutter & Siskind classic "Lambda the ultimate backpropagator" [TOPLAS'08]

Existential + value-dependent types to the rescue

```
T1 => T2 =
   exists Δ. (x : T1) -> Σ (y : T2). G[y : T2] -> (Δ, G[x : T1])

G [ v : T1 => T2 ] =
   case v of
   | exists Δ _ => Δ
```

Δ : cotangent space of the environment over which we closed over.

Note that it's also _returned_ by the pullback!

```
curry :: ((T,S) => R) -> (T => S => R)
curry (exists D. f) = pack () new_f
  where new_f :: (t:T) -> ((g : S => R), G[g:S=>R] -> (D, G[t:T])
        new_f t =
           let g :: (s:S) -> (r:R, G[r:R] -> ((D,G[t:T]), G[s:S])
               g s =
                  let (r, pullback) = f(t,s)
                  in (r, \gr -> let (cte,(ctt,cts)) = pullback gr
                                in ((cte,ctt), cts))
              new_pb :: G[g:S=>R] -> (D, G[t:T])
              new_pb env = env    // Magic (but type-correct)!
           in (pack (D,G[t:T]) g, new_pb)
```

Have a formalization of this idea in dependent type theory (Agda)

Plus proofs of the CCC laws in Coq. Tricky bits:

- Precise notion of equivalence
    - Requires a higher-dimensional LR
- Encoding issues in a theorem prover (avoid large eliminations, use of recursion-recursion, another talk really)

But Swift is not dependently-typed …

Efficient solution: no recomputation but with reinterpret casts (AnyDerivative object)

```
curry :: ((T,S) => R) -> (T => (S => R))
curry (exists D. f) = pack () new_f
  where
    new_f :: (t:T) -> ((g : S => R), G[g:S=>R] -> (D, G[t:T])
    new_f t =
       let g :: (s:S) -> (r:R, G[r:R] -> ((D,G[t:T]), G[s:S])
           g s =
              let (r, pullback) = f(t,s)
              in (r, \gr -> let (cte,(ctt,cts)) = pullback gr
                                in ((cte,ctt), cts))
           new_pb :: G[g:S=>R] -> (D, G[t:T])
           new_pb env = env    // Magic (but type-correct)!
       in (pack [..] g, new_pb)
```

```
G[S => T] = AnyDerivative // An “opaque” type with 0 and +
S => T = (S -> (T, G[T] -> (AnyDerivative,G[S])))

curry :: ((T,S) => R) -> (T => (S => R))
curry ([redacted]f) = [redacted]new_f
                                                        AnyDerivative
  where
   new_f :: ([redacted]T) -> (([redacted]S => R), G[[redacted]S=>R] -> (D, G[[redacted]T])
   new_f t =
      let g :: ([redacted]S) -> ([redacted]R, G[[redacted]R] -> ((D,G[t:T]), G[[redacted]S])
          g s =
                                                      AnyDerivative
            let (r, pullback) = f(t,s)
            in (r, \gr -> let (cte,(ctt,cts)) = pullback gr
                              in ((cte,ctt), cts))
          new_pb :: G[[redacted]S=>R] -> (D, G[[redacted]T])
          new_pb env = env
      in ([redacted]g, new_pb)           AnyDerivative
```

Is the type-erased solution workable?

- Main appeal: efficiency and simplicity
- Main disadvantage: AnyDerivative **not safe** without storing runtime information. Hence it can be used *internally* by the AD compiler pass but not exposed to users. Practically this means:

$$\nvdash \texttt{Differentiable (A => B)}$$

Hence, can't write an external function and make it return gradients to "f"

```
func my_own_curry(f : (T,S) => R, x:T) : S => R  = \s -> f(x,s)
func bar(y : U) =  my_own_curry (\(t,s) -> … use y here …), … )
// Can't make "bar" differentiable, though if we had inlined everything, it'd all work!
```

Higher order derivatives: a sketch

```
newtype a => b = DiffRec (a -> (b, Bundle (G[a]) (G[b])))

data Bundle ta tb
  = BundleEnd
  | BundleTan { deriv :: (AnyDer, ta) -> (tb, Bundle ta tb),
                pullback :: tb -> (AnyDer, ta, Bundle ta ta) }
vjp :: (a => b) -> Grad b -> (a => Grad a)
vjp (DiffRec f) gb = DiffRec new_f
  where new_f :: a -> (G[a], Bundle (G[a]) (G[a]))
        new_f a = let (b, bundle :: Bundle (Grad a) (Grad b)) = f a
                  in case bundle of
                     BundleEnd -> BundleEnd
                     BundleTan deriv pullback ->
                           // Just pick the inner bundle!
                           let (any, ta, b) = pullback gb in (ta, b)
// Similarly for jvp:
jvp :: (a => b) -> Grad a -> (a => Grad b)
jvp = …
```

We need, ahead of time, to create a data structure that's amenable to arbitrary differentiation. That is, we need to make the user-facing vjp and jvp return themselves differentiable functions

Allows to compute, e.g., Hessian-vector products:

```
hvp(f, primals, vs) =
   jvp(\x -> vjp f 1.0) vs primals
```

Idea can be extended with "constantly zero" derivatives, infinitely unfoldable bundles (e.g. for sin() and exp()) etc.

Thoughts and outlook

- In better shape if AD had happened after explicit closure conversion?
- "Safe" AnyDerivative through runtime tests? Or by using the "slow" `G[T=>R] = List (T, G[R])` for functional arguments, but internally fall back to the "fast" dependently-typed (erased) version?
- Control flow, recursion, recursive types => know how to deal with, orthogonal

Slides covered just a fragment of the much more complete language-based AD design behind S4TF, including experimental extensions and discussion. More S4TF questions (including AD)? Reach out:

swift@tensorflow.org

Thanks!

- Forward/reverse mode symbolic AD is fairly simple in A-normal form or SSA
- Simple AD rules + differentiable curry = AD for HO functions
- Simple AD rules + recursive bundle structure = HO AD

A new interpretation of function gradients plus some simplification and proofs of ideas behind *"Lambda the Ultimate Backpropagator"* in a statically typed setting