A tale of two cities

McCoy R. Becker

Friday, April 24, 2020

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to heaven, we were all going direct the other way - in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

Charles Dickens, 1859

This is a modified version of a discussion I gave last week to the compiler group. I've slimmed this version down, and made it slightly more opinionated.

> *The expression of automatic differentiation and probabilistic programming systems rely on a similar set of programming patterns. These can be identified by thinking about the 'run-times' of these systems and what representations they work with. In particular, you'll find much in common between these systems and compilers/interpreters.*

This is survey oriented.

Modern AI

- Automatic differentiation
- Probabilistic programming
- Discrete optimization

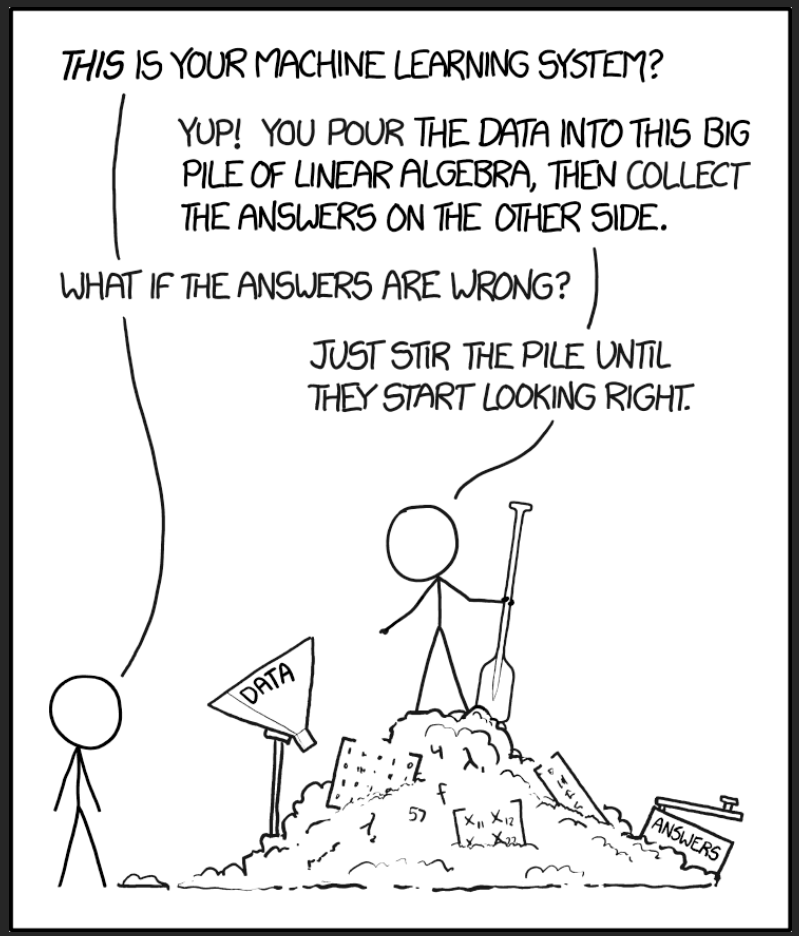
This is essentially the whole story. So why aren't we *there* yet?

---

* *there* is artificial general intelligence.

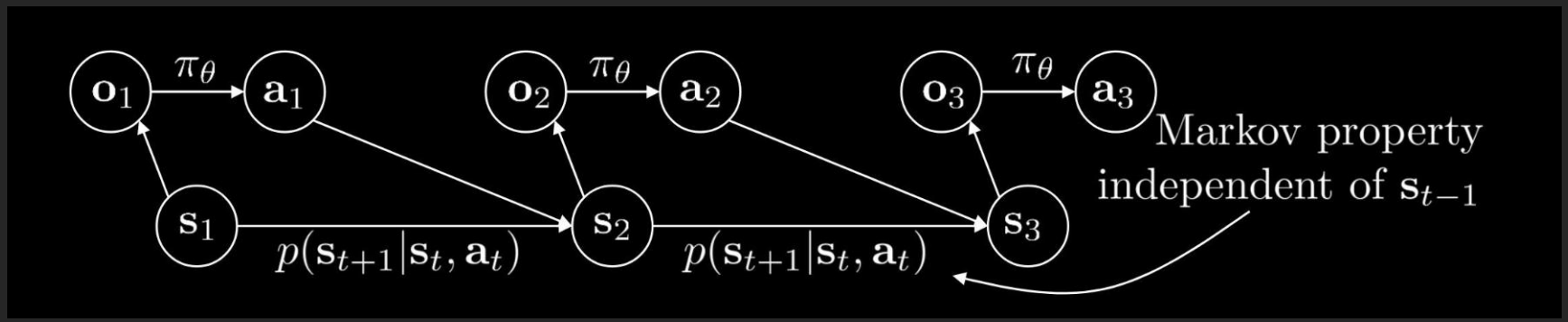
Stuck at local optima (in abstraction space)?

- Deep learning: core abstraction is the computation graph, which represents the flow of differentiable information in your program. You fill the graph up with tensors. Deep learning engineers are paid large amounts to build these graphs.



Stuck at local optima (in abstraction space)?

- Probabilistic programming: core abstraction is either a form of dataflow graph (i.e. an explicit network representation of a model) or a dynamic structure representing a program trace, depending on the probabilistic programming system.



In the flow graph case, the information content of the graph is different compared to deep learning - the connections represent the conditional dependencies.

Long tale short

There are graph representations, and there are dynamic “program trace representations” which perform the correct analysis and computation during runtime (by i.e. collecting information specific to that execution).

A static graph representation is amenable to analysis (like a compiler) but programming to that representation is more restrictive. This is often required to achieve the absolute best performance (see XLA).

The dynamic representation usually allows you to program directly in the host language.

There is no single “best” way to combine deep learning and PP...

- Sampling from a distribution is not directly differentiable [1]. When including sampling, the correct thing is to compute the expected loss over the sampling distribution.
- Static, graph based representations of probabilistic programs allow for restricted use of ”black box” operations (i.e. I want to shove a deep network in the middle of my PP).

  This then restricts what inference algorithms you can use.
- How do you differentiate a probabilistic program [2]? Can you apply reparametrization tricks to a PP?

---

[1] This requires the usage of *reparametrization tricks* which produce gradient estimators.

[2] Semantically, this is understood as: you differentiate the log probability of the data. But this is not trivial to acquire for all models.

A modern tower of Babel

Can you really blame these communities (and the implementors of frameworks) for not talking to each other?

It requires tremendous experience to become high-level in any one of these areas. Additionally, they care about performance in different ways:

- Deep learning: must deploy to heterogeneous accelerators and many GPUs.
- PP: complex proposals require high CPU power - some exact algorithms are just matrix muls. More important in general to thus parallelize across many threads of a CPU.

Let's take a closer look at the abstractions to identify the microcosm of the main issue: improvement to "mutual abstractions" is not a high priority.

Automatic differentiation

```
function foo(x::Float64, y::Float64)
    return sin(exp(x) + y)
end
```

Simple - but the core ideas apply to complex differentiable programs.

The chain rule of differential calculus

$$\frac{d}{dx} f(g(h(x))) = \left(\frac{df}{df}\bigg|_{f(x)}\right) \times \left(\frac{df}{dg}\bigg|_{g(x)}\right) \times \left(\frac{dg}{dh}\bigg|_{h(x)}\right) \times \left(\frac{dh}{dx}\bigg|_{x}\right) \quad (1)$$

- Forward mode: start on the right hand side.
- Reverse mode: start on the left hand side.

Forward mode

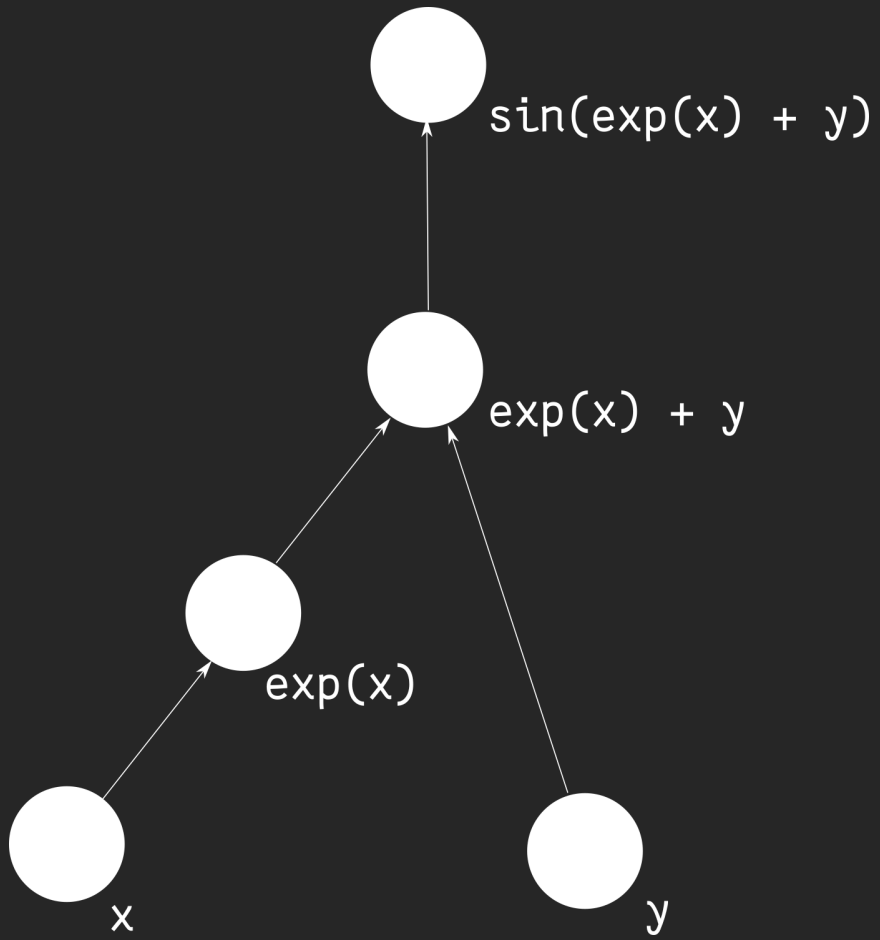


Figure 1: A computation graph.

$$x = x \Longrightarrow dx = dx$$

$$y = y \Longrightarrow dy = dy$$

$$y_1 = exp(x) \Longrightarrow dy_1 = exp(x)\, dx$$

$$y_2 = y_1 + y \Longrightarrow dy_2 = dy_1 + dy$$

$$y_3 = sin(y_2) \Longrightarrow dy_3 = cos(y_2)\, dy_2$$

Reverse mode

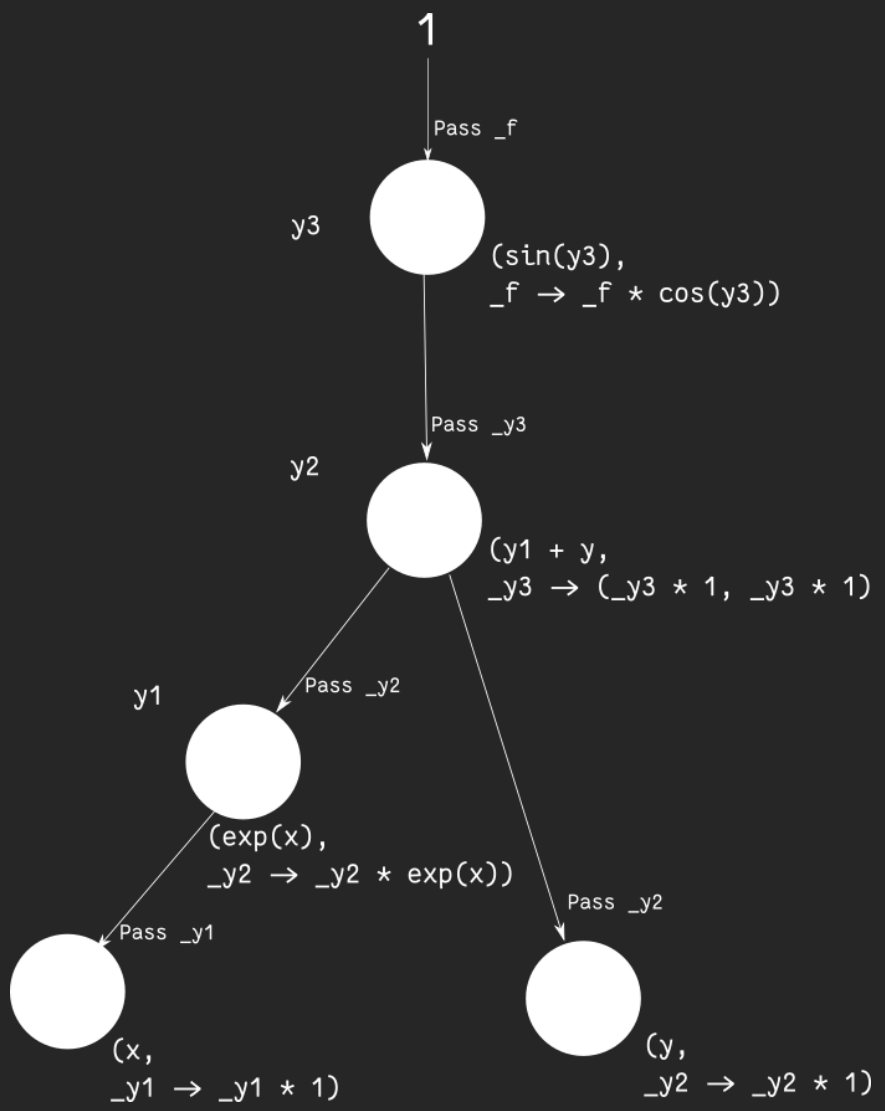


Figure 2: An adjoint computation graph.

The core representation is a “computation graph” which represents the flow of *differentiable* information through the program.

Depending on the implementation, this graph may be represented explicitly before execution (i.e. TF 1) or implicitly by execution (PyTorch and TF 2/Eager).

Probabilistic programming

```
function foo(x::Float32)
    y = rand(Normal(x, 1.0))
    z = rand(Normal(y, 1.0))
    return z
end
```

The key operation for probabilistic programming is *inference*.

$$P(y|z) = \frac{P(z|y)P(y)}{P(z)} \qquad \text{(Bayes)}$$

Given observed data z, I want to update the distribution over y to reflect the data.

Forms of inference	Requires
Sampling algorithms	Ability to sample from program
Exact inference with conjugate families	Ability to reason about primitive distributions in program
Variational inference	In black box version, the ability to sample from program
Belief propagation	Ability to reason about primitive distributions in program

Static (graph-based) PPLs

See *Figaro*, for example.

Here, the model is constructed by explicitly connecting the graph together *before* runtime.

- Advantage: appears to be the closest to the computation graph abstraction.
- Advantage: a static representation allows analysis (as in every compiler ever).
- Disadvantage: requires that you learn a DSL.
- Disadvantage: Not necessarily compatible with other interpretations (i.e. AD) - depending on model, might be difficult to compute log prob.

Trace-based PPLs

```
function foo(x::Float32)
    res = Array{Float64, 1}([])
    while rand(Normal(0.0, 1.0)) < 3.0
        push!(res, 1)
    end
    if length(res) > 10
        y = rand(Normal(10.0, 5.0))
    else
        y = rand(Normal(5.0, 3.0))
    end
    return y
end
```

Key insight to understanding: running this program samples from a distribution over execution traces. What measurable space is that distribution defined on?

The choice map abstraction

```
@gen function foo(x::Float32)
    res = Array{Float64, 1}([])
    counter = 0
    # :x => counter is addr
    while @trace(normal(0.0, 1.0),
                 :x => counter) < 1.0
        push!(res, 5)
        counter += 1
    end
    return res
end
```

We explicitly represent the trace by storing addresses and random choices.

Here's a sample choicemap:

```
└── :x
    ├── 0 : -0.0646385063500121 7
    ├── 4 : -2.3258112470761123
    ├── 2 : -0.42855543766951
    ├── 3 : -0.5157659627666945
    ├── 5 : 1.199858833348223 2
    └── 1 : 0.18879462594409707
```

Not shown: accumulated log probabilities which go into the total log probability for this particular choice map.

Monte Carlo sampling engines

Importance sampling

$$\mathbb{E}_{z\sim P(z|x)}[f(z)] = \int f(z)\ P(z|x)\ dz$$

$$= \int f(z)\ \underbrace{\frac{P(z|x)}{Q(z)}\ Q(z)}_{\text{Requires Q a.c. wrt P}}\ dz$$

$$\approx (\sum_{i=0}^{N} w_i\ f(z_i))/(\sum_{i=0}^{N} w_i) \quad \text{(Monte Carlo estimate)}$$

$$w_i = \frac{P(x|z_i)P(z_i)}{Q(z_i)} \quad \text{(Importance weights)}$$

Programmable inference

In trace-based sampling methods, you write down a proposal *program* $Q(z)$.

The only constraint is that the program has to have the same support as the unobserved variables in the original one.

Note that this follows from the absolute continuity requirement given on the previous slide.

- Advantage: You can construct highly complex proposal distributions for sampling algorithms.
- Advantage: Gradient information always available using trace-based auto diff.
- Disadvantage: no static analysis - sampling always happens at runtime. You miss out on things like e.g. identifying conjugacy.
- Disadvantage: numerous exact and approximate inference algorithms don't fit well into this framework (i.e. belief propagation based algorithms).
- Disadvantage: auto diff *must be* trace-based. You can't do source-to-source AD if you only know the log probability at runtime.

A summary of representation issues

Here's the crux - language issues exist between fields...but they also exist *within fields*!

Deep learning:

- Limited support for complex architectures (e.g. TreeRNN, capsule networks, neural program policies) - i.e. the things we should be trying.
- True *differentiable programming* where everything has an adjoint.

Probabilistic programming:

- Loss of useful inference information at runtime.
- Static frameworks seen as "less flexible" compared to trace-based.

These issues have always existed between static and dynamic systems.

They arise from the fundamental tension between what can be determined by the compiler and what cannot be determined until data is flowing.

Machine learning people $\neq$ compiler people

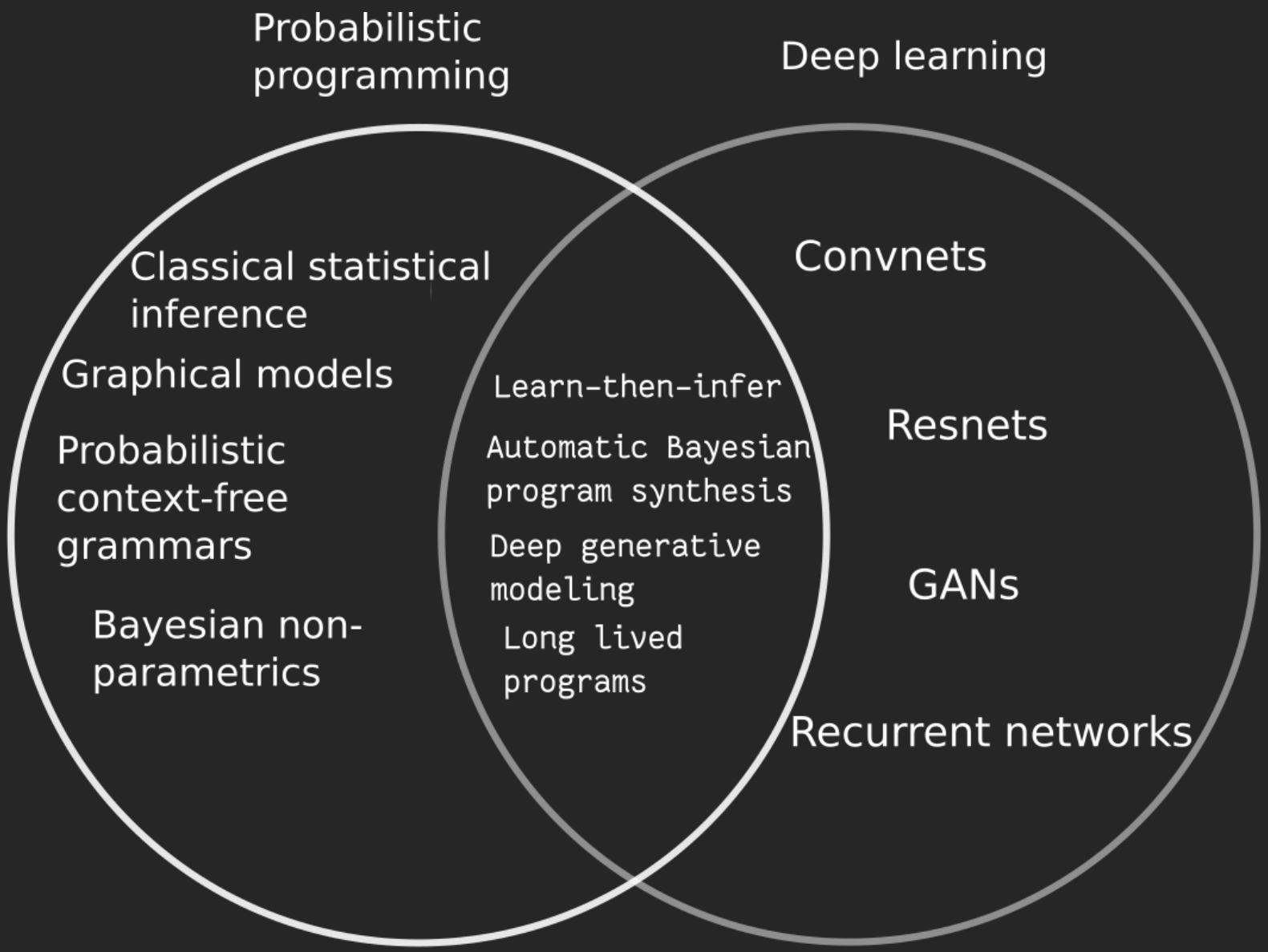
...

but they are starting to learn about compilers.

Combining ‘contexts’

What about AD and probabilistic programming all in one?

- This is a language problem.
  1. What subset of your language supports probabilistic programming? What subset of your language is differentiable? Can they be made to intersect?
- Design space:
  1. Use trace based for everything? Poor performance on complex architectures. Think: pure interpreter approach.
  2. Source-to-source? No existing source-to-source PPL package which is “universal” - initial work done on ”density compilers” (see *Hakaru*). Ongoing work on *Jaynes*.


Probabilistic programming
Deep learning
Classical statistical inference
Graphical models
Probabilistic context-free grammars
Bayesian non-parametrics
Learn-then-infer
Automatic Bayesian program synthesis
Deep generative modeling
Long lived programs
Convnets
Resnets
GANs
Recurrent networks

The future...

Many bleeding edge efforts restrict themselves to a purely functional subset of a host language...

- JAX - functional subset of Python (yikes!)
- NumPyro - effect handlers in Python (yikes!)
- Possibly XLA... (not in Python)

Pure functional languages are easy to reason about because mutability is controlled and composition makes flow-of-control very explicit.

In other words, everything is local...can we make AD + PP local as well?

Algebraic effects

A principled way to write programs with “semantic interception”.

If you’ve ever used exception handling, you’ve used an effect.

Let’s say you write some code, and you want to imbue it with a “context” where certain calls are handled in a particular way.

```
-- The effect
ability Differentiable where
    cos : Float -> Float
    sin : Float -> Float
    exp : Float -> Float

Tape : List (Float -> Float)
Tape = []

-- A handler - in charge of handling the effect in the context.
pullback : List (Float -> Float) -> Request Differentiable a -> List (Float -> Float)
pullback l = cases
    { Differentiable.cos x -> k} -> handle
                                        k (Float.cos x)
                                    with pullback (l List.++ [y -> -1.0 * y * Float.sin x])

    { Differentiable.exp x -> k} -> handle
                                        k (Float.exp x)
                                    with pullback (l List.++ [y -> y * Float.exp x])

    { Differentiable.sin x -> k} -> handle
                                        k (Float.sin x)
                                    with pullback (l List.++ [y -> y * Float.cos x])

    { x } -> l List.++ [y -> y * 1.0]

gradient_tape : (Float -> Float) -> Float -> List (Float -> Float)
gradient_tape v x =
    handle v x with pullback []

grad : (Float -> Float) -> Float -> Float
grad f x = (foldl (x -> y -> y x) 1.0) (gradient_tape f x)


-- Test!
> grad Differentiable.exp 5.0
> grad Differentiable.sin 0.0
> grad Differentiable.cos 0.0

p : Float -> Float
p x = Differentiable.exp (Differentiable.cos x)

> grad p 10.0
```

So a program gets handled in a context...the handler can express many forms of computation.

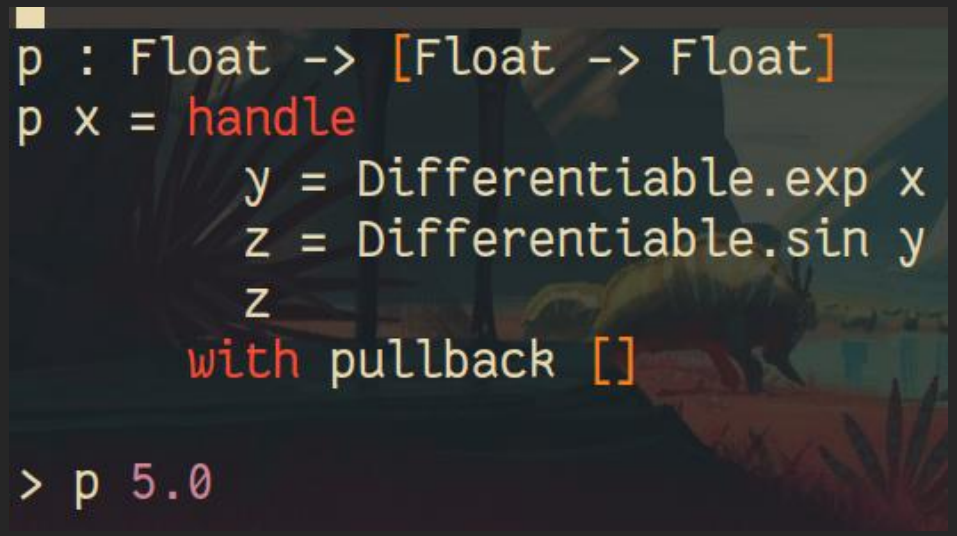
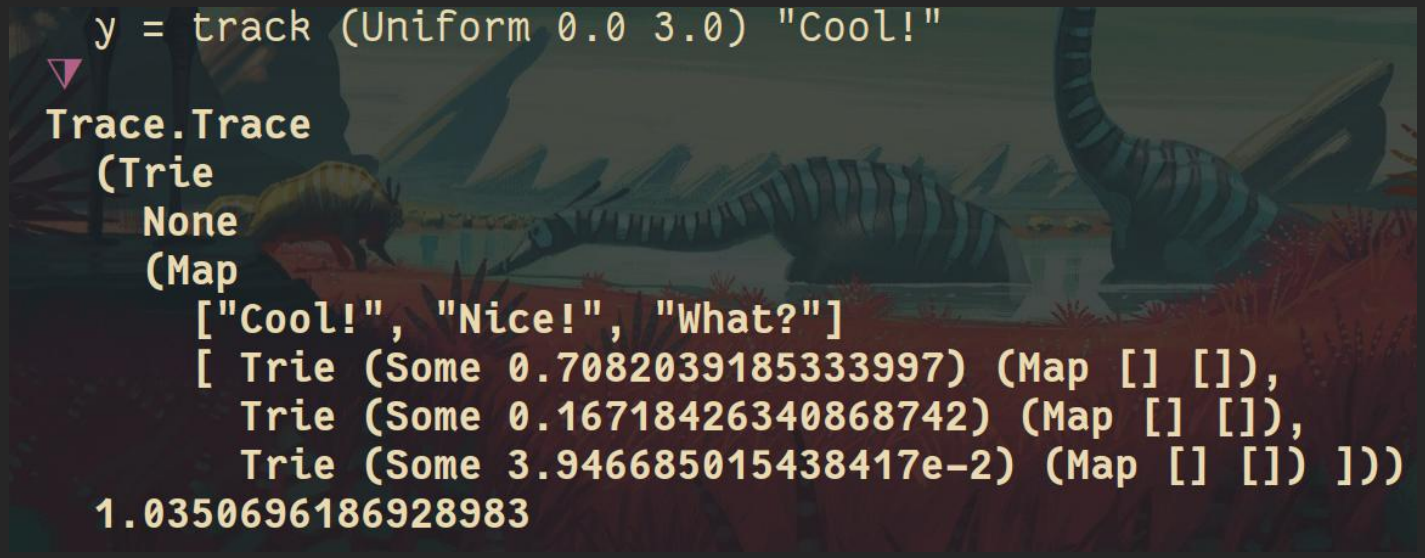


Figure 3: More explicit: any call which requires the *Differentiable* ability in the context is handled explicitly by a specific handler.

Why restrict this to AD? Why not PP?

(I’m glad you asked…)


-- Distributions
type Distribution = Uniform Float Float
sample : Distribution ->{Randomness} Float
sample = cases
Uniform x y -> y - x * rand
logprob : a -> Distribution -> Float
logprob a = cases
Uniform x y ->
use Float - /
log (1.0 / (y - x))
-- Tracing (i.e. choice maps)
type Trace k v = Trace (Trie k v) Float
ability Tracing where
track : a -> Text ->{Tracing} Float
record : k -> v -> Distribution -> Trace k v -> Trace k v
record txt val dist tr =
recorded_val_tr = choice_map.modify (Trie.insert [txt] val) tr
logp = logprob val dist
new_tr =
logprob.set (Trace.logprob recorded_val_tr + logp) recorded_val_tr
new_tr
trace : Trace Text Float -> Request Tracing Float -> Trace Text Float
trace tr = cases
{ Tracing.track call addr -> k } ->
x = handle Randomness.sample call with lcg +0
new_tr = record addr x call tr
handle k x with trace new_tr
{ x } -> tr
-- Probabilistic programming!
> handle
y = track (Uniform 0.0 3.0) "Cool!"
z = track (Uniform 0.0 y) "Nice!"
q = track (Uniform z (z + 5.0)) "What!"
q
with trace tr


y = track (Uniform 0.0 3.0) "Cool!"
Trace.Trace
(Trie
None
(Map
["Cool!", "Nice!", "What?"]
[Trie (Some 0.70820391853339 97) (Map [] []),
Trie (Some 0.1671842634086874 2) (Map [] []),
Trie (Some 3.946685015438417e-2) (Map [] [])]))
1.0350696186928983

Why not both *at the same time*?

(Again, wonderful question inquisitive listener...)

Effects are a really wonderful idea.

- Functional way (i.e. composable!) way to handle side effects.
- Can be type checked!

They allow you to explicitly control and check for *context* with the type system!

In *Scruff* development, we have a notion called *model capabilities* which I believe is modelled by functional effect systems.

Summary

At their cores, deep learning and probabilistic programming systems are very similar. However, what information is required to compute the thing of interest, as well as how it is computed are very different.

An interesting avenue for research: unified representations for machine learning systems which allow the flexibility to express complex differentiable models with probabilistic ones.