Design Doc: funsor Trace_ELBO [WIP]

Authors: Yerdos Ordabayev

First edit: 2021-10-10 - Last edit: 2021-11-05

[Objective](#)

[Related Work](#)

[Design Overview](#)

[Detailed Design](#)
- [Fine-grained Rao-Blackwellization](#)
  - [Example 1](#)
  - [Example 2](#)
  - [Using SumProduct and Adjoint algorithms to obtain Rao-Blackwellized measures](#)
    - [Example 1](#)
    - [Example 2](#)
    - [Extended Example 1 - multiple cost terms](#)
- [Dependency (provenance) tracking](#)
- [Dice factors as importance weights](#)
    - [Separability](#)
- [WIP](#)

Objective

Create a general version of `Trace_ELBO` that combines `Trace_ELBO`, `TraceGraph_ELBO`, and `TraceEnum_ELBO`.

$$ELBO = E_{q(z)}[\log p(x, z) - \log q(z)] = E_{q(z)}[f(z)]$$

where $f(z)$ represents cost terms. The objective is to obtain an unbiased gradient estimator that is infinitely differentiable -- construct ELBO in such a way that the derivative of any order can be pushed through the expectation:

$$\nabla^n ELBO = \nabla^n E_{q(z)}[\dots] = E_{q(z)}\left[\nabla^n \dots\right]$$

Related Work

- pyro.Trace_ELBO is implemented along the lines of Automated Variational Inference in Probabilistic Programming and Black Box Variational Inference. There are no restrictions on the dependency structure of the model or the guide. The gradient estimator includes partial Rao-Blackwellization for reducing the variance of the estimator when non-reparameterizable random variables are present. The Rao-Blackwellization is partial in that it only uses conditional independence information that is marked by plate contexts.
- pyro.TraceGraph is implemented along the lines of reference Gradient Estimation Using Stochastic Computation Graphs specialized to the case of the ELBO. It supports arbitrary dependency structure for the model and guide as well as baselines for non-reparameterizable random variables. Where possible, conditional dependency information as recorded in the Trace is used to reduce the variance of the gradient estimator. In particular two kinds of conditional dependency information are used to reduce variance: 1) the sequential order of samples (z is sampled after y => y does not depend on z) and 2) plate generators.
- pyro.TraceEnum_ELBO that supports exhaustive enumeration over discrete sample sites, and - local parallel sampling over any sample site in the guide. This assumes restricted dependency structure on the model and guide: variables outside of a plate can never depend on variables inside that plate.
- First- and Second-Order Expectation Semirings with Applications to Minimum-Risk Training on Translation Forests -- shows relationship between expectations and gradients. This led to the idea of using constant identity terms (targets) to determine log measures for cost terms.
- Monte Carlo Gradient Estimation in Machine Learning -- nice overview of Monte Carlo Gradient estimation
- DiCE: The Infinitely Differentiable Monte Carlo Estimator and Storchastic: A Framework for General Stochastic Automatic Differentiation -- papers about Dice factor.
- ...

Design Overview

1. Fine-grained Rao-Blackwellization. Use variable elimination (`sum_product`) and adjoint algorithms to efficiently determine `log_measures` for each cost term -- same approach is already used for enumerated sites in `contrib.funsor.TraceEnum_ELBO` pyro#2725
2. Dependency tracking for *non-enumerated* sites is achieved by using the `funsor.torch.ProvenanceTensor` funsor#543
3. For this approach to work Dice factors are treated as importance weights. Importance sampling is represented by an `Importance` funsor which 1) eagerly evaluates to `Delta+dice_factor` and 2) is normalized when reduced.

Detailed Design

Fine-grained Rao-Blackwellization

Rao-Blackwellization allows variance reduction in Monte Carlo estimators. This is achieved by calculating expected value of each cost term *only* over stochastic nodes that cost term depends on (accounting for plate conditional independence as well). Consider these two examples:

Example 1

In this example the cost term $log\ p(c)$ depends only on "c" and thus it's Rao-Blackwellized measure is the marginal distribution $q(c) \equiv \int_b q(b)\ q(c \mid b)$.

```
# a     b --> c --> d
# cost term - log_p(c)
```

$$E_{q(a,b,c,d)}[log\ p(c)] = \int_{a,b,c,d} log\ p(c)\ q(a)\ q(b)\ q(c \mid b)\ q(d \mid c)$$

$$= \int_c log\ p(c) \int_b q(b)\ q(c \mid b) \int_{a,d} q(a)\ q(d|c)$$

$$= \int_c log\ p(c) \int_b q(b)\ q(c|b)$$

Example 2

In this example each $log\ p(z_i)$ cost term depends only on z_i and therefore the Rao-Blackwellized measure is the marginal distribution $q(z_i)$.

```
# +----------+
# | q(z_i)   |
# |       N |
# +----------+
# cost terms - log_p(z_i)
```

$$E_{q(z)}\left[log\ p(z_i) \right] = \int_{z_i} log\ p(z_i) \prod_i^N q(z_i)$$

$$= \int_{z_i} log\ p(z_i)\ q(z_i) \int_{z_j} \prod_{j \neq i}^N q(z_j)$$

$$= \int_{z_i} log\ p(z_i)\ q(z_i)$$

From which it follows that:

$$E_{q(z)}\left[\sum_i^N log\ p(z_i)\right] = \sum_i^N \int_{z_i} log\ p(z_i)\ q(z_i)$$

Using *SumProduct* and *Adjoint* algorithms to obtain Rao-Blackwellized measures

This is implemented in [pyro#2893]

Note that the marginal measure of a cost term is its adjoint and can be determined in three steps:

1. For each cost term, create a *target*, a constant identity term of the same shape.
2. Apply `sum_product` variable elimination algorithm on all targets and all measures (this returns a normalization factor Z)
3. Apply the `adjoint` algorithm on Z wrt targets.

Example 1

$$\int_{a,b,c,d} 1(c)\ q(a)\ q(b)\ q(c \mid b)\ q(d \mid c)\ =\ Z \equiv 1$$

$$\nabla_{1(c)} Z = \nabla_{1(c)} \int_c 1(c)\ q(c)\ = q(c) \equiv \int_b q(b)\ q(c \mid b)$$

Example 2

$$\int_{z_i} 1(z_i) \prod_i q(z_i) = Z \equiv 1$$

$$\nabla_{1(z_i)} Z = \nabla_{1(z_i)} \int_{z_i} 1(z_i)\ q(z_i) = q(z_i)$$

Extended Example 1 - multiple cost terms

```
# new cost terms: log_q(a), log_q(b), log_q(c|b), log_p(d|c)
# new targets:    1(a),     1(b),     1(b,c),     1(c,d)
```

$$\int_{a,b,c,d} 1(a)\ 1(b)\ 1(c)\ 1(b,c)\ 1(c,d)\ q(a)\ q(b)\ q(c \mid b)\ q(d \mid c) = Z$$

$$\nabla_{1(a)} Z = q(a)$$

$$\nabla_{1(b)} Z = q(b)$$

$$\nabla_{1(c)} Z = q(c) \equiv \int_b q(b)\ q(c \mid b)$$

$$\nabla_{1(b,c)} Z = q(b,c) \equiv q(b)\ q(c \mid b)$$

$$\nabla_{1(c,d)} Z = q(c,d) \equiv q(c)\ q(d \mid c)$$

The benefit of using *targets* is that they can be shared between different cost terms and the `sum_product+adjoint` algorithms need to be run only once. In principle, this approach should work for all delayed/eager sampling and enumeration approaches and for all funsor probability distributions: `Distribution`, `Delta`, `Tensor`, `Gaussian`. Constant identity terms (targets) are represented by a `Constant(cost.inputs, Number(0))` funsor (funsor#548).

For the `sum_product+adjoint` approach described above to work it is important that distributions q are normalized.

Dependency (provenance) tracking

Dependency tracking automatically works with delayed sampling and enumeration strategies. Here we deal with the dependency (provenance) tracking of eagerly sampled non-enumerated sites. Proposed approach is to use `ProvenanceTensor` (funsor#543). E.g., in example 1 `log_p(c)` depends on “c” -- if site “c” is not enumerated then we can use ProvenanceTensor to track dependency:

```
# c is a ProvenanceTensor(value, provenance={“c”})
c = pyro.sample(“c”, ...)
# log_prob_c is ProvenanceTensor(log_prob_value, provenance={“c”})
log_prob_c = dist.log_prob(c)
```

Two rules for provenance tracking:

1. `pyro.sample(name, ...)` returns `ProvenanceTensor(value, provenance={name})`
2. The provenance of the output tensor is the union of provenances of input tensors:

```
a = ProvenanceTensor(torch.tensor(3.), {“a”})
b = ProvenanceTensor(torch.tensor(2.), {“b”})
c = a + b
assert c._provenance == a._provenance | b._provenance
```

`ProvenanceTensor` is converted to `Constant` funsor since their behaviours are very similar. `ProvenanceTensor._provenance` corresponds to `Constant.const_inputs`:

```
>>> to_funsor(c)
Constant(OrderedDict(a=Real, b=Bint[3]), Tensor(torch.tensor(5.))
```

Dice factor as an importance weight

Derivatives of any order can be pushed through the expectation if the distribution can be reparameterized or exactly integrated via the enumeration. Any order derivatives of non-reparameterizable distributions are handled via the introduction of Dice factors.

A Dice factor $\frac{q(z)}{\bar{q}(z)}$ can be viewed as an importance weight where the proposal distribution $\bar{q}(z)$ is the same distribution but with detached gradients:

$$E_{q(z)}[\, f(z)\,] = E_{\bar{q}(z)}\left[\, \frac{q(z)}{\bar{q}(z)} f(z)\,\right]$$

This allows to push the gradient straight through the expectation to get:

$$\nabla E_{\bar{q}(z)}\left[\, \frac{q(z)}{\bar{q}(z)} f(z)\,\right] = E_{\bar{q}(z)}\left[\nabla \frac{q(z)}{\bar{q}(z)} f(z)\,\right] = E_{\bar{q}(z)}\left[\, \frac{\nabla q(z)}{\bar{q}(z)} f(z) + \frac{q(z)}{\bar{q}(z)} \nabla f(z)\,\right]$$

Since `Delta(name, point) + dice_factor` is not normalized (since gradients in `dice_factor` are non-zero) and therefore we take the approach of returning `Delta(name, point, dice_factor)` instead as explained below.

Let’s consider integral below:

$$\int_{a,b} 1(b)\, \bar{q}(a) \frac{q(a)}{\bar{q}(a)}\, \bar{q}(b) \frac{q(b)}{\bar{q}(b)} = Z \equiv 1$$

When $\bar{q}(b)$ and $\bar{q}(b)$ are eagerly sampled the integral above becomes:

$$\int_{a,b} 1(b)\,\bar{\delta}(a{:}\,a_s)\frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}\,\bar{\delta}(b{:}\,b_s)\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)} = \frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}\,\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)}$$

where $\bar{\delta}(a{:}\,a_s)\frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}$ and $\bar{\delta}(b{:}\,b_s)\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)}$ are `Delta + dice_factor`'s. The problem is that this integral is not normalized! The proposed solution is to represent `Delta + dice_factor` as a single funsor `Importance(model_delta, guide_delta)` and exploit the equality:

$$\int_a \left[\bar{\delta}(a{:}\,a_s)\frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}\right] = \int_a \delta(a{:}\,a_s) = 1$$

In the example above:

$$\int_{a,b} 1(b)\left[\bar{\delta}(a{:}\,a_s)\frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}\right]\left[\bar{\delta}(b{:}\,b_s)\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)}\right] = \int_b 1(b)\left[\bar{\delta}(b{:}\,b_s)\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)}\right]\int_a\left[\bar{\delta}(a{:}\,a_s)\frac{\delta(a=a_s)}{\bar{\delta}(a=a_s)}\right]$$

$$= \int_b\left[\bar{\delta}(b{:}\,b_s)\frac{\delta(b=b_s)}{\bar{\delta}(b=b_s)}\right] = 1$$

```
Importance(model_delta, guide_delta).reduce(ops.logaddexp, name) ==
model_delta.reduce(ops.logaddexp, name)
```

Importance funsor is implemented in [funsor#578]:

1. Signature - `Importance(model, guide, sampled_vars)`
2. When guide is a `Delta` it eagerly evaluates to `guide + model - guide`
3. `Importance.reduce` is delegated to `Importance.model.reduce`

When model = Delta(name, point, log_prob) and guide = Delta(name, point, ops.detach(log_prob)):

```
guide + model - guide
      == guide + log_prob - ops.detach(log_prob)
      == guide + dice_factor
```

The last peace is lazy_interpretation for Importance

```
lazy_importance = DispatchedInterpretation("lazy_importance")

@lazy_importance.register(Importance, Funsor, Delta, frozenset)
def _lazy_importance(model, guide, sampled_vars):
    return reflect.interpret(Importance, model, guide, sampled_vars)
```

It is used for a lazy importance sampling:

```
with lazy_importance:
      sampled_dist = dist.sample(msg["name"], sample_inputs)
```

and for adjoint algorithm:

```
with lazy_importance:
      marginals = adjoint(funsor.ops.logaddexp, funsor.ops.add, logzq)
```

Distributivity

```
model_a = Delta(“a”, point_a[“i”], log_prob_a)
guide_a = Delta(“a”, point_a[“i”], ops.detach(log_prob_a))
q_a = Importance(model_a, guide_a, {“a”})

model_b = Delta(“b”, point_b[“i”], log_prob_b)
guide_b = Delta(“b”, point_b[“i”], ops.detach(log_prob_b))
q_b = Importance(model_b, guide_b, {“b”})

with lazy_importance:
      (q_a.exp() * q_b.exp() * cost_b).reduce(add, {“a”, “b”, “i”})
      == [q_a.exp().reduce(add, “a”) * (q_b.exp() * cost_b).reduce(add,
{“b”})].reduce(add, “i”)
      == [1(“i”) * (q_b.exp().reduce(add, {“b”}) +
cost_b(b=point_b))].reduce(add, “i”)
      == [1(“i”) * cost_b(b=point_b)].reduce(add, “i”)
      == cost_b(b=point_b).reduce(add, “i”)
```

WIP

```
# Example 0
# a     b --> c
# cost term - log_p(c) - has a dependency on “c”
```

$E_{q(a,b,c)}[log\ p(c)] = \int_{a,b,c} q(a)\ q(b)\ q(c \mid b)\ log\ p(c) = [\int_{a} q(a)]\ [\int_{b,c} q(b)\ q(c \mid b)\ log\ p(c)]$

Integrate(q_a + q_b + q_c, cost_c, {"a", "b", "c"})
 --montecarlo→ Integrate(

```
                        Delta({"a": …}) + Tensor(dice_a)
                        + Delta({"b": …}) + Tensor(dice_b)
                        + Delta({"c": …}) + Tensor(dice_c),
                       cost_c, {"a", "b", "c"})
--rao-blackwell→
```

ImportanceWeight funsor rules:

Lemma 1) Integrate(Delta(“a”: point_a) + ImportanceWeight(dice_a, {"a"}), cost_a, {“a”}) ==
        ==
        == (dice_a.exp() * cost_a(a=point_a)).reduce(ops.add, {“a”})

Lemma 2) Integrate(Delta(“a”: point_a) + ImportanceWeight(dice_a, {"a"}), cost_b, {“a”}) == cost_b

Axiom 1) ImportanceWeight(dice_a, {"a"}).reduce(ops.logaddexp, {“a”}) = 0 # normalized

Axiom 2) Delta(“a”: point_a) + ImportanceWeight(dice_a, {"a"}) ≠ Delta(“a”: point_a) + ImportanceWeight(dice_a, {"a"})(a=point_a)

Axiom 3) Integrate(Delta({"a": …}) + ImportanceWeight(dice_a, {"a"}), cost_a, {"a"})
  = (dice_a + cost_a).reduce(ops.logaddexp, {a"})
this combines the integrate-delta rule with the delta-constant rule

Addition rules:

f is any funsor except Delta and Constant
(Importance(..., “a”) + f(“a”)).reduce(logaddexp, “a”)
        == (Delta(“a”) + log_importance_weight + f(a=point_a)).reduce(logaddexp, “a”)
Delta(“a”) - will this case even arise?
(Importance(..., “a”) + Delta(“a”)).reduce(logaddexp, “a”)
        ==
Constant
(Importance(..., “a”) + Constant(“a”, 0)).reduce(logaddexp, “a”)
        == Importance(..., “a”).reduce(logaddexp, “a”)
        == model.reduce(logaddexp, “a”) = 0

(Importance(..., “a”) + f(“b”)).reduce(ops.logaddexp, “a”)
        == Importance(..., “a”).reduce(ops.logaddexp, “a”) + f(“b”) = 0 + f(“b”)

(f(“x”, “i”) * g(“y”, “i”)).reduce(add, {“x”,”y”,”i”}) ==
   (f(“x”, “i”).reduce(add, “x”) * g(“y”, “i”).reduce(add, “y”)).reduce(add, “i”)
   != (f(“x”, “i”).reduce(add, {“x”, “i”}) * g(“y”, “i”).reduce(add, {“y”, “i”}))

Integrate(

```
   log_measure=Delta("x", point) + Normalized(df, wrt="x"),
   integrand=Term("x"),
   reduced_var=”x”
)
```

is equivalent to:

```
(Delta("x", point) + Normalized(df, wrt="x") + Term("x")).reduce(add, "x")
        => (Delta("x", point) + Normalized(df, wrt="x") + Term("x")(x=point)).reduce(add, "x")


        => (Delta("x", point)).reduce(logaddexp, "x") + n + Term("x")(x=point)
        => n + Term("x")(x=point)


(Delta("x", point) + Normalized(n, wrt="x") + Number(3.0)).reduce(logaddexp, "x")
        => (Delta("x", point) + Normalized(n, wrt="x")).reduce(logaddexp, "x") + Number(3.0)


Delta("x", point) + Normalized(dice_factor, wrt="x") + Term("x")(x=point)

Delta("x", point) + Normalized(dice_factor, wrt="x") + Annotated(Term("x")(x=point),
depends_on=”x”)  ??
```

Integrate(
Normalized(Delta(“x”, point)+DiceFactor, wrt=”x”),
Variable(“x”),
“x”
)
=> (Delta(“x”, point) + DiceFactor + Variable(“x”)).reduce(“x)

Integrate(
Normalized(Delta(“x”, point)+DiceFactor, wrt=”x”),
Number(3),
“x”
)
=> Normalized(Delta(“x”, point)+DiceFactor, wrt=”x”).reduce(“x) + Number(3)
=> 0 + Number(3)

```
Integrate(
        Normalized(Delta(“x”, point)*DiceFactor, wrt=”x”),
```

```
        Constant(3, wrt=x),
        “x”
)
 => Normalized(Delta(“x”, point)*DiceFactor, wrt=”x”) * Constant(3, wrt=x)  .reduce(“x)
 => Delta(x) * DiceFactor * Constant(3, wrt=x)  .reduce(x)
 => 3 * DiceFactor

Don’t merge Normalized!
if lhs.wrt.isdisjoint(rhs.wrt):  # ??
Normalized(f, wrt=”x”) + Normalized(g, wrt=”y”) = Normalized(f+g, wrt={“x”, “y”})

Integrate(
        Normalized(Delta(“x”, point)+Delta(“y”, point)+DFx+DFy, wrt={”x”, “y”}),
        Variable(“x”),
        “xy”
)
 => (Delta(“x”, point) + DiceFactor + Variable(“x”)).reduce(“x)

(Normalized(Delta + DiceFactor, wrt=”x”) + Constant(0, wrt=”x”)).reduce(“x”)
 => Delta + DiceFactor + 0 => DiceFactor

q(a)q(b|a)q(c|b)f(c) .reduce(a,b,c) =
Normalized(Delta(a)*DF_a, wrt=a) * Normalized(Delta(b)*Const(DF_b, wrt=a), wrt=b) *
Normalized(Delta(c)*Const(DF_c, wrt=b), wrt=c) * F(c)  .reduce(a, b, c)

Delta(a) * DF_a * Normalized(Delta(b)*Const(DF_b, wrt=a), wrt=b) *
Normalized(Delta(c)*Const(DF_c, wrt=b), wrt=c) * F(c)  .reduce(a, b, c)

DF_a * Normalized(Delta(b)*DF_b, wrt=b) * Normalized(Delta(c)*Const(DF_c, wrt=b), wrt=c) *
F(c)  .reduce(b, c)

DF_a * DF_b * Normalized(Delta(c)*DF_c, wrt=c) * F(c)  .reduce(c)

DF_a * DF_b * DF_c * F(c=point_c)

DF_a * DF_b * F(b=point_b)


q(a) q(b|a) q(c|b) 1(a) 1(b,a) 1(c,b) .reduce(a,b,c) = Z

adjoint(Z, 1(a)) = q(a)

1(a) * q(a) .reduce(a) = Z
adjoint of 1(a) = > q(a)
```

Norm(q(a), wrt=a) * Norm(q(b|a), wrt=b,a) * Norm(q(c|b), wrt=c,b) * Norm(1(a), wrt=a)
Norm(1(b,a), wrt=b,a) * Norm(1(c,b), wrt=b,c) .reduce(a,b,c)
=>
=> 1(a) * q(a) * DF_b * DF_c .reduce(a) = Z
adjoint of 1(a) = > q(a) # * DF_b * DF_c

```
Integrate(
        Normalized(Delta("x", point)+DiceFactor, wrt="x"),
        Constant(3),
        "x"
)
```

Normalized(q(a), wrt=a) * Normalized(1(a), wrt=a) .reduce(a) => 1

#####
q(a) q(b|a) 1(a) 1(b,a) .reduce(a,b) = Z

Normalized(Delta(a)*DF_a, wrt=a) * Normalized(Delta(b)*Const(DF_b, wrt=a), wrt=b) * 1(a) *
1(b, a) .reduce(a, b)

Normalized(Delta(a)*DF_a, wrt=a) * Delta(b) * Const(DF_b, wrt=a) * 1(a) * 1(b, a) .reduce(a, b)

Normalized(Delta(a)*DF_a, wrt=a) * Const(DF_b, wrt=a) * 1(a) .reduce(a)

Delta(a) * DF_a * Const(DF_b, wrt=a) * 1(a) .reduce(a)

Delta(a) * DF_a * DF_b .reduce(a)

DF_a * DF_b

Adjoint of 1(a):

Delta(a) * DF_a * DF_b

Integrate(p(a), 1(a), a) = 1

Integrate(q(a), 1(a), a) =