Functional tensors for probabilistic programming

Fritz Obermeyer, Eli Bingham, Martin Jankowiak,
Du Phan, JP Chen (Uber AI)

NeurIPS workshop on program transformation 2019-12-14

Outline

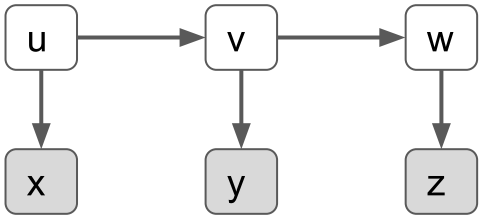
Motivation

What are Funsors?

Language overview

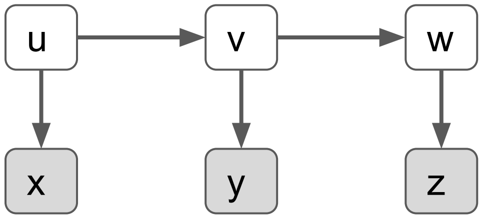
Discrete latent variable models

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Categorical(F[0])
v ~ Categorical(F[u])
w ~ Categorical(F[v])
observe x ~ Categorical(H[u])
observe y ~ Categorical(H[v])
observe z ~ Categorical(H[w])
```



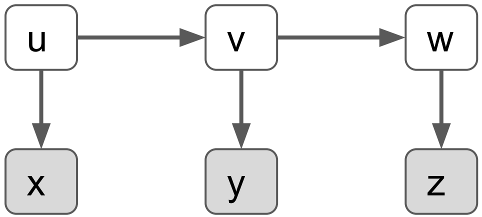
Discrete latent variable models

```
F = pyro.param("F", torch.ones(n,n), constraint=simplex)
H = pyro.param("H", torch.ones(n,m), constraint=simplex)
u = pyro.sample("u", Categorical(F[0]))
v = pyro.sample("v", Categorical(F[u]))
w = pyro.sample("w", Categorical(F[v]))
pyro.sample("x", Categorical(H[x]), obs=x)
pyro.sample("y", Categorical(H[y]), obs=y)
pyro.sample("z", Categorical(H[z]), obs=z)
```



Discrete latent variable models

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Categorical(F[0])
v ~ Categorical(F[u])
w ~ Categorical(F[v])
observe x ~ Categorical(H[u])
observe y ~ Categorical(H[v])
observe z ~ Categorical(H[w])
```



Inference via variable elimination

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Categorical(F[0])
v ~ Categorical(F[u])
w ~ Categorical(F[v])
observe x ~ Categorical(H[u])
observe y ~ Categorical(H[v])
observe z ~ Categorical(H[w])
```

Goal: vary F,H to maximize p(x,y,z)

$$p(x, y, z) = \sum_u \sum_v \sum_w p(u, v, w, x, y, z)$$

Inference via variable elimination

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Categorical(F[0])
v ~ Categorical(F[u])
w ~ Categorical(F[v])
observe x ~ Categorical(H[u])
observe y ~ Categorical(H[v])
observe z ~ Categorical(H[w])
```

Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \sum_u \sum_v \sum_w p(u,v,w,x,y,z)$$

$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

Inference via variable elimination

Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \sum_u \sum_v \sum_w p(u,v,w,x,y,z)$$

$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

Inference via variable elimination

```
# In a named tensor library:
p = (F(0,"u")*F("u","v")*F("v","w")
      *H("u",x)*H("v",y)*H("w",z)
      ).sum("u").sum("v").sum("z")
```

Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \sum_u \sum_v \sum_w p(u,v,w,x,y,z)$$

$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

Inference via variable elimination

```
# In a named tensor library:
p = (F(0,"u")*F("u","v")*F("v","w")
      *H("u",x)*H("v",y)*H("w",z)
      ).sum("u").sum("v").sum("z")
```

Cost is exponential in # variables

Goal: vary F,H to maximize p(x,y,z)

$$p(x, y, z) = \sum_u \sum_v \sum_w p(u, v, w, x, y, z)$$

$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

Inference via variable elimination

```
# In a named tensor library:
p = (F(0,"u")*F("u","v")*F("v","w")
  *H("u",x)*H("v",y)*H("w",z)
  ).sum("u").sum("v").sum("z")
```

Cost is exponential in # variables

Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \sum_u \sum_v \sum_w p(u,v,w,x,y,z)$$

$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

$$= \sum_u F_{0,u} H_{u,x} \sum_v F_{u,v} H_{v,y} \sum_w F_{v,w} H_{w,z}$$

Cost is linear in # variables

Inference via variable elimination

```
# In a named tensor library:
p = (F(0,"u")*F("u","v")*F("v","w")
     *H("u",x)*H("v",y)*H("w",z)
     ).sum("u").sum("v").sum("z")
```

```
# In PyTorch:
p = einsum("u,vu,vw,u,v,w",
           F[0],F,F,
           H[:,x],H[:,y],H[:,z])
```

```
p.backward()  # backprop to optimize F,H
```

Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \sum_u \sum_v \sum_w p(u,v,w,x,y,z)$$

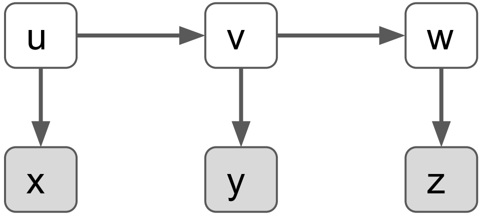
$$= \sum_u \sum_v \sum_w F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}$$

$$= \sum_u F_{0,u} H_{u,x} \sum_v F_{u,v} H_{v,y} \sum_w F_{v,w} H_{w,z}$$

Cost is linear in # variables

~~Discrete~~ Gaussian latent variable models

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Normal(0,1)
v ~ Normal(u,1)
w ~ Normal(v,1)
observe x ~ Normal(u,1)
observe y ~ Normal(v,1)
observe z ~ Normal(w,1)
```



Kalman filters,
Sequential Gaussian Processes,
Linear-Gaussian state space models,
Gaussian conditional random fields,
...

~~Discrete~~ Gaussian latent variable models

```
F : Tensor[n,n]
H : Tensor[n,m]
u ~ Normal(0,1)
v ~ Normal(u,1)
w ~ Normal(v,1)
observe x ~ Normal(u,1)
observe y ~ Normal(v,1)
observe z ~ Normal(w,1)
```

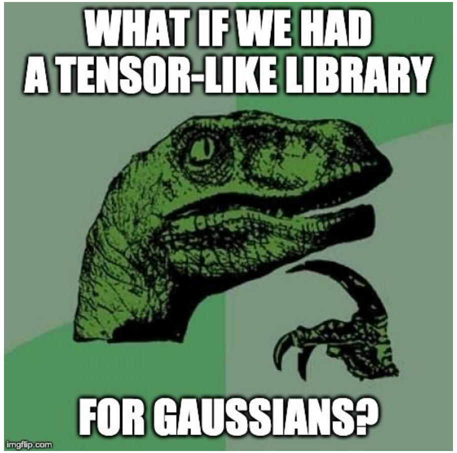
Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \iiint p(u,v,w,x,y,z)\, du\, dv\, dw$$

$$= \iiint F_{0,u} F_{u,v} F_{v,w} H_{u,x} H_{v,y} H_{w,z}\, du\, dv\, dw$$

$$= \int F_{0,u} H_{u,x} \int F_{u,v} H_{v,y} \int F_{v,w} H_{w,z}\, du\, dv\, dw$$

~~Discrete~~ Gaussian latent variable models



Goal: vary F,H to maximize p(x,y,z)

$$p(x,y,z) = \iiint p(u,v,w,x,y,z)\,du\,dv\,dw$$

$$= \iiint F_{0,u}F_{u,v}F_{v,w}H_{u,x}H_{v,y}H_{w,z}\,du\,dv\,dw$$

$$= \int F_{0,u}H_{u,x} \int F_{u,v}H_{v,y} \int F_{v,w}H_{w,z}\,du\,dv\,dw$$

```
# In a gaussian library:
p = (F(0,"u")*F("v","u")*F("v","w")
     *H("u",x)*H("v",y)*H("w",z)
     ).sum("u").sum("v").sum("z")  # or .integrate() or something?
```

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)

"Tensors are open terms
whose dimensions are free variables
of type bounded int"

"Funsors are open terms
whose free variables are
of type bounded int or real array"

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)
- A Gaussian over multiple variables is still Gaussian (i.e. higher rank)

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)
- A Gaussian over multiple variables is still Gaussian (i.e. higher rank)
- We still need integer dimensions for batching
- We still need discrete Tensors for e.g. Gaussian mixtures

Funsor ::= Tensor | Gaussian | ...

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)
- A Gaussian over multiple variables is still Gaussian (i.e. higher rank)
- We still need integer dimensions for batching
- We still need discrete Tensors for e.g. Gaussian mixtures
- Gaussians are closed under some operations:
  - Gaussian * Gaussian ⇒ Gaussian
  - Gaussian.sum("a_real_variable") ⇒ Gaussian
  - Gaussian["x" = affine_function("y")] ⇒ Gaussian
  - (Gaussian * quadratic_function("x")).sum("x") ⇒ Gaussian or Tensor

How can we compute with Gaussians?

- Tensor dimensions → free variables (real-valued or vector-valued)
- A Gaussian over multiple variables is still Gaussian (i.e. higher rank)
- We still need integer dimensions for batching
- We still need discrete Tensors for e.g. Gaussian mixtures
- Gaussians are closed under some operations:
  - Gaussian * Gaussian ⇒ Gaussian
  - Gaussian.sum("a_real_variable") ⇒ Gaussian
  - Gaussian["x" = affine_function("y")] ⇒ Gaussian
  - (Gaussian * quadratic_function("x")).sum("x") ⇒ Gaussian or Tensor
- Gaussians are not closed under all operations:
  - Gaussian.sum("an_integer_variable") ⇒ ...a mixture of Gaussians…
  - (Gaussian * f("x")).sum("x") ⇒ ...an arbitrary Gaussian expectation...

Funsors are not as simple as Tensors

Approximate computation with Gaussians

```
Gaussian.sum("i") ⇒ ...mixture of Gaussians...
# but approximating...
with interpretation(moment_matching):
    Gaussian.sum("i") ⇒ Gaussian
```

But nonstandard interpretation helps!

Approximate computation with Gaussians

```
Gaussian.sum("i") ⇒ ...mixture of Gaussians...
# but approximating...
with interpretation(moment_matching):
    Gaussian.sum("i") ⇒ Gaussian
```

```
(Gaussian * f("x")).sum("x") ⇒ ...arbitrary expectation...
# but approximating...
with interpretation(monte_carlo):
    (Gaussian * f("x")).sum("x") ⇒ Gaussian or Tensor
```

But nonstandard interpretation helps!

Approximate computation with Gaussians

```
Gaussian.sum("i") ⇒ ...mixture of Gaussians...
# but approximating...
with interpretation(moment_matching):
    Gaussian.sum("i") ⇒ Gaussian
```

```
(Gaussian * f("x")).sum("x") ⇒ ...arbitrary expectation...
# but approximating...
with interpretation(monte_carlo):
    (Gaussian * f("x")).sum("x") ⇒ Gaussian or Tensor
```

a randomized rewrite rule

But nonstandard interpretation helps!

Monte Carlo approximation via Delta funsors

```
# Three rewrite rules:
with interpretation(monte_carlo):
    (Gaussian * f("x")).sum("x") ⇒ (Delta * f("x")).sum("x")

Delta("x",x,w) * f("x") ⇒ Delta("x",x,w) * f(x)

Delta("x",x,w).sum("x") ⇒ w
```

Monte Carlo approximation via Delta funsors

```
# Three rewrite rules:
with interpretation(monte_carlo):
    (Gaussian * f("x")).sum("x") ⇒ (Delta * f("x")).sum("x")

Delta("x",x,w) * f("x") ⇒ Delta("x",x,w) * f(x)

Delta("x",x,w).sum("x") ⇒ w
```

The point x and weight w are both differentiable:
- x via the reparameterization trick,
- w via REINFORCE, DiCE factor
  (e.g. to track mixture component weight)

Monte Carlo approximation via Delta funsors

```
# Three rewrite rules:
with interpretation(monte_carlo):
    (Gaussian * f("x")).sum("x") ⇒ (Delta * f("x")).sum("x")

Delta("x",x,w) * f("x") ⇒ Delta("x",x,w) * f(x)

Delta("x",x,w).sum("x") ⇒ w
```

The point x and weight w are both differentiable:
- x via the reparameterization trick,
- w via REINFORCE, DiCE factor

Theorem: monte_carlo is correct in expectation at all derivatives.

Inference via delayed sampling

fun $\mathrm{GenerativeModel}(x)$	$p \leftarrow 1$
$\quad z \leftarrow \mathrm{sample}(P_z)$	$p \leftarrow p \times P_z[v = z]$
$\quad y \leftarrow \exp(z)$	
$\quad \mathrm{observe}(P_x[\theta = y],\ x)$	$p \leftarrow p \times P_x[\theta = y, v = x]$
end	maximize: $\sum_z p$

Funsor syntax

```
Funsor ::= Tensor | Gaussian | Delta | Variable
         | Funsor["x"=Funsor]              # substitution
         | f̂(Funsor, …, Funsor)             # application, e.g. +,*
         | ∑ₓ Funsor                        # marginalization
         | ∏ₓ Funsor                        # plate reduction
```

Algorithm 1 TensorVariableElimination

input variabl… F, values $F \subseteq V \times F$,

plate s…

plate s…

output PlatedSumProduct(…

Initialize an empty list of scalars $S \leftarrow []$.

while F is not empty **do**

- Choose a leaf plate set $L \in \{P(f) \mid f \in F\}$ with a maximal number of plates.
- Let $V_L \leftarrow \{v \in V \mid P(v) = L\}$ be the variables in L.
- Let $F_L \leftarrow \{f \in F \mid P(f) = L\}$ be the factors in L.
- Let $E_L \leftarrow E \cap (V_L \times F_L)$ be the edges in L.
- **for** (V_c, F_c) **in** Partition(V_L, F_L, E_L) **do**
  - Let $f \leftarrow$ SumProduct(F_c, V_c).
  - Let $V_f \leftarrow \{v \mid (v, f) \in E \cap ((V \setminus V_c) \times F_c)\}$ be the set of f's remaining variables.
  - Remove component (V_c, F_c) from V, F, E, P.
  - **if** V_f is empty **then**
    - Add Product(f, L, M) to scalars S.
  - **else**
    - Let $L' \leftarrow \bigcup\{P(v) \mid v \in V_f\}$ be the next plate set where f has variables.
    - **if** $L' = L$ **then error**("Intractable!");
    - Let $f' \leftarrow$ Product$(f, L \setminus L', M)$.
    - Add f' to F, E, P appropriately.

return SumProduct$(S, \{\})$

```
def plated_sum_product(sum_op, prod_op, factors, eliminate, plates):
    sum_vars = eliminate - plates
    ...
    scalars = []
    while ordinal_to_factors:
        leaf = max(ordinal_to_factors, key=len)
        leaf_factors = ordinal_to_factors.pop(leaf)
        leaf_vars = ordinal_to_vars[leaf]
        for (group_factors, group_vars) in partition(leaf_factors, leaf_vars):
            f = reduce(prod_op, group_factors).reduce(sum_op, group_vars)
            remaining_sum_vars = sum_vars.intersection(f.inputs)
            if not remaining_sum_vars:
                scalars.append(f.reduce(prod_op, leaf & eliminate))
            else:
                new_plates = frozenset().union(
                    *(var_to_ordinal[v] for v in remaining_sum_vars))
                if new_plates == leaf:
                    raise ValueError("Intractable!")
                f = f.reduce(prod_op, leaf - new_plates)
                ordinal_to_factors[new_plates].append(f)
    return reduce(prod_op, scalars)
```

This would have been heinously complex without Funsors

Questions?

github.com / pyro-ppl / funsor ← code

funsor.pyro.ai ← docs

arxiv.org / abs / 1910.10775 ← longer paper

Extra Material

Variational inference

fun GenerativeModel(x)	$p \leftarrow 1$
$\quad z \leftarrow \text{sample}(P_z)$	$p \leftarrow p \times P_z[v = z]$
$\quad \text{observe}(P_x[\theta = z],\ x)$	$p \leftarrow p \times P_x[v = x, \theta = z]$
end	
fun InferenceModel(x)	$q \leftarrow 1$
$\quad z \leftarrow \text{sample}(Q[\theta = x])$	$q \leftarrow q \times Q[v = z, \theta = x]$
end	maximize: $\sum_z q \log \frac{p}{q}$

Pyro as modeling frontend

A new DSL for inference backend

modeling frontend

```
def model():
    x = pyro.sample("x", Px)
    y = pyro.sample("y", Py(θ=x),
                    obs=data)
```

Pyro

inference backend

```
p = 1
p *= Px(x="x")
p *= Py(θ="x")(y=data)


p = p.sum()  # marginalize out x
loss = -log(p)
loss.backward()
```

PSEUDOCODE

modeling frontend

```
def guide(data):
    x = pyro.sample("x", Qx(data))

def model(data):
    x = pyro.sample("x", Px)
    y = pyro.sample("y", Py(θ=x),
                          obs=data)
```

Pyro

inference backend

```
log_q = 0
log_q += Qx(data)(x="x")

log_p = 0
log_p += Px(x="x")
log_p += Py(θ="x")(y=data)

elbo = log_q.exp() * (log_p - log_q)
elbo = elbo.sum()  # marginalize out x
loss = -elbo
loss.backward()
```

PSEUDOCODE

modeling frontend

semi-symbolic backend

```
y = Tensor(torch.randn(10))
assert isinstance(y + y, Tensor)   # eager

x = Variable("x", reals(10))
assert isinstance(x + x, Binary)   # lazy
assert isinstance(x + y, Binary)   # lazy
```

Funsor

Pyro

```
from pyro.generic import distributions as dist
from pyro.generic import infer, optim, pyro, pyro_backend

def model(data):
    locs = pyro.param("locs", torch.tensor([-1., 0., 1.]))
    with pyro.plate("plate", len(data), dim=-1):
        x = pyro.sample("x", dist.Categorical(torch.ones(3) / 3))
        pyro.sample("obs", dist.Normal(locs[x], 1.), obs=data)

def guide(data):
    with pyro.plate("plate", len(data), dim=-1):
        p = pyro.param("p", torch.ones(len(data), 3) / 3, event_dim=1)
        pyro.sample("x", dist.Categorical(p))

for backend in ["pyro", "funsor"]:
    with pyro_backend(backend):
        svi = infer.SVI(model,  guide,  optim.Adam({}),  infer.Trace_ELBO())
        svi.step(data=torch.randn(10))
```

Uses funsor under the hood

```python
from pyro.generic import distributions as dist
from pyro.generic import infer, optim, pyro, pyro_backend

def model(data):
    locs = pyro.param("locs", torch.tensor([-1., 0., 1.]))
    with pyro.plate("plate", len(data), dim=-1):
        x = pyro.sample("x", dist.Categorical(torch.ones(3) / 3))
        pyro.sample("obs", dist.Normal(locs[x], 1.), obs=data)

def guide(data):
    with pyro.plate("plate", len(data), dim=-1):
        p = pyro.param("p", torch.ones(len(data), 3) / 3, event_dim=1)
        pyro.sample("x", dist.Categorical(p))

for backend in ["pyro", "funsor"]:
    with pyro_backend(backend):
        svi = infer.SVI(model,  guide,  optim.Adam({}),  infer.Trace_ELBO())
        svi.step(data=torch.randn(10))
```

Pyro

Uses funsor under the hood

Pyro

```
from pyro.generic import distributions as dist
from pyro.generic import infer, optim, pyro, pyro_backend

def model(data):
    locs = pyro.param("locs", torch.tensor([-1., 0., 1.]))
    with pyro.plate("plate", len(data), dim=-1):
        x = pyro.sample("x", dist.Categorical(torch.ones(3) / 3))
        pyro.sample("obs", dist.Normal(locs[x], 1.), obs=data)

def guide(data):
    with pyro.plate("plate", len(data), dim=-1):
        p = pyro.param("p", torch.ones(len(data), 3) / 3, event_dim=1)
        pyro.sample("x", dist.Categorical(p))

for backend in ["pyro", "funsor"]:
    with pyro_backend(backend):
        svi = infer.SVI(model,  guide,  optim.Adam({}),  infer.Trace_ELBO())
        svi.step(data=torch.randn(10))
```

Uses funsor under the hood

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())     # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)      # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)     # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)           # emission

    return log_p
```

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())      # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)       # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)          # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)             # emission

    return log_p
```

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())       # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)          # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)             # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)                # emission

    return log_p
```

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())     # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)       # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)          # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)             # emission

    return log_p
```

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())        # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)           # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)              # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)                 # emission

    return log_p
```

Funsor

```
def kalman_filter_model(data):
    log_p = 0.
    x_curr = funsor.Tensor(torch.tensor(0.))

    for t, y in enumerate(data):
        x_prev = x_curr
        x_curr = funsor.Variable('x_{}'.format(t), funsor.reals())       # delayed sample
        log_p += dist.Normal(x_prev, trans_noise, value=x_curr)          # transition
        if isinstance(x_prev, funsor.Variable):
            log_p = log_p.reduce(ops.logaddexp, x_prev.name)             # eagerly collapse prev state
        log_p += dist.Normal(x_curr, emit_noise, value=y)                # emission

    return log_p
```

PyTorch

```python
class Encoder(nn.Module):
  def __init__(self):
    super(Encoder, self).__init__()
    self.fc1 = nn.Linear(784, 400)
    self.fc21 = nn.Linear(400, 20)
    self.fc22 = nn.Linear(400, 20)
  def forward(self, image):
    image = image.reshape(
      image.shape[:-2] + (-1,))
    h1 = F.relu(self.fc1(image))
    loc = self.fc21(h1)
    scale = self.fc22(h1).exp()
    return loc, scale

class Decoder(nn.Module):   . . .
```

Funsor

```python
encode = funsor.torch.function(
  reals(28, 28), (reals(20), reals(20)))(Encoder())

decode = funsor.torch.function(
  reals(20), reals(28, 28))(Decoder())

@funsor.interpretation(funsor.monte_carlo)
def vae_loss(data):
  loc, scale = encode(data)
  q = funsor.Independent(
        dist.Normal(loc['i'], scale['i'], value='z'), 'z', 'i')

  probs = decode('z')
  p = dist.Bernoulli(probs['x', 'y'], value=data['x', 'y'])
  p = p.reduce(ops.add, frozenset(['x', 'y']))

  elbo = funsor.Integrate(q, p - q, frozenset(['z']))
  return -elbo.reduce(ops.add, 'batch')
```

Funsor

image : reals(28,28) ⊢ encode : reals(20) × reals(20)

z : reals(20) ⊢ decode : reals(28,28)

```
encode = funsor.torch.function(
    reals(28, 28), (reals(20), reals(20)))(Encoder())

decode = funsor.torch.function(
    reals(20), reals(28, 28))(Decoder())

@funsor.interpretation(funsor.monte_carlo)
def vae_loss(data):
    loc, scale = encode(data)
    q = funsor.Independent(
            dist.Normal(loc['i'], scale['i'], value='z'), 'z', 'i')

    probs = decode('z')
    p = dist.Bernoulli(probs['x', 'y'], value=data['x', 'y'])
    p = p.reduce(ops.add, frozenset(['x', 'y']))

    elbo = funsor.Integrate(q, p - q, frozenset(['z']))
    return -elbo.reduce(ops.add, 'batch')
```