Funsor: Functional Tensors for Probabilistic Programming

Fritz Obermeyer*, Eli Bingham*, Martin Jankowiak*, JP Chen, Du Phan

Introduction: We need ‘autograd for integrals’

Probabilistic modelling and inference offer **a unifying approach to many machine learning tasks**, including quantifying uncertainty, learning structured generative models, producing interpretable explanations of data, and learning from weak or missing labels.

Probabilistic programming languages like Pyro allow **specification of probabilistic models in high-level programming languages.**

But many models that mix many different discrete and continuous variables still need custom inference strategies, and there is **no lower-level analogue of automatic differentiation software** intermediate between fully symbolic and fully black box integration.

We extend lazy tensor expressions to include **dimensions of size ‘real’** and implement **semisymbolic integration**

On GitHub: pyro-ppl/funsor

Functional Tensors: a language for automatic integration over array-valued variables

Probabilistic programs generate lazy expressions with free variables. Inference algorithms integrate over free variables:

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

Approximate inference computations also generate lazy sum-product expressions in the same expression language.

$$
\begin{array}{l|l}
\textbf{fun}\ \mathrm{GenerativeModel}(x) & p \leftarrow 1 \\
\quad z \leftarrow \mathrm{sample}(P_z) & p \leftarrow p \times P_z[v=z] \\
\quad \mathrm{observe}(P_x[\theta=z],\ x) & p \leftarrow p \times P_x[v=x, \theta=z] \\
\textbf{end} & \\
\textbf{fun}\ \mathrm{InferenceModel}(x) & q \leftarrow 1 \\
\quad z \leftarrow \mathrm{sample}(Q[\theta=x]) & q \leftarrow q \times Q[v=z, \theta=x] \\
\textbf{end} & \mathrm{maximize:}\ \sum_z q \log(p/q)
\end{array}
$$

Pyro represents terms with discrete free variables as torch.Tensors. We extend this representation to other functions, encoding them as “tensors” where some of the “dimensions” have size “real”:

$$
\begin{array}{rll}
\tau \in \mathrm{Type} ::= & \mathbb{Z}_n & \text{“bounded integer”} \\
| & \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k} \to \mathbb{R} & \text{“real-valued array”}
\end{array}
$$

We define a set of specific terms that are closed under variable substitution, sum-product operations, and various approximations:

$$
\begin{array}{rll}
e \in \mathrm{Funsor} ::= & \mathrm{Tensor}(\Gamma, w) & \text{“discrete factor”} \\
| & \mathrm{Gaussian}(\Gamma, i, P) & \text{“Gaussian factor”} \\
| & \mathrm{Delta}(v, e) & \text{“point mass”} \\
| & \mathrm{Variable}(v, \tau) & \text{“delayed value”} \\
| & \hat{f}(e_1, \ldots, e_n) & \text{“apply function”} \\
| & e_1[v=e_2] & \text{“substitute”} \\
| & \sum_v e & \text{“marginalize”} \\
| & \prod_{v/s} e & \text{“Markov product”}
\end{array}
$$

Approximation and transformation via (re-)interepretation

Most integrals cannot be computed directly and must be simplified. We rewrite lazy expressions by evaluating them with many different interpreters.

Some rules trigger PyTorch ops:

```
@eager.register (Binary, Op, Tensor, Tensor)
def eager_binary_tensor(op, lhs, rhs):
  inputs, (x, y) = align_tensors(lhs, rhs)
  data = op(x.data, y.data)
  return Tensor(data, inputs, lhs.dtype)
```

Some trigger further rewrites:

```
@eager.register(Binary, AddOp, Delta, Funsor)
def eager_add_delta(op, lhs, rhs):
  if lhs.name in rhs.inputs:
    rhs = rhs(**{lhs.name: lhs.point})
    return op(lhs, rhs)
  return None  # defer to default implementation
```

Some rewrite subexpressions into approximate versions. `monte_carlo` rewrites Tensor and Gaussian to Delta:

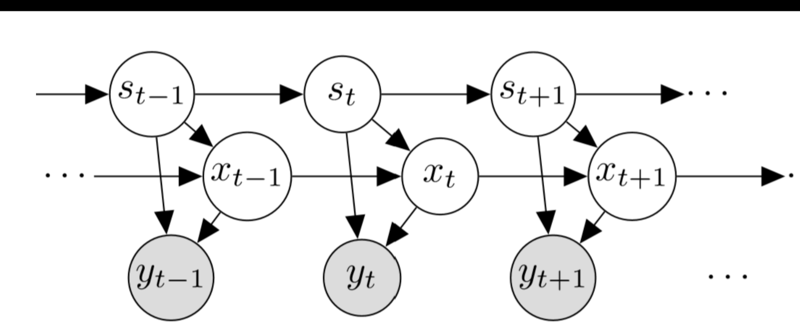
```
@dispatched_interpretation
def monte_carlo(cls, *args): ...

@monte_carlo.register(Integrate, Funsor, Funsor, set)
def monte_carlo_integrate(log_measure, integrand, vs):
  log_measure = log_measure.sample()
  return eager.dispatch(
    Integrate, log_measure, integrand, vs)
```

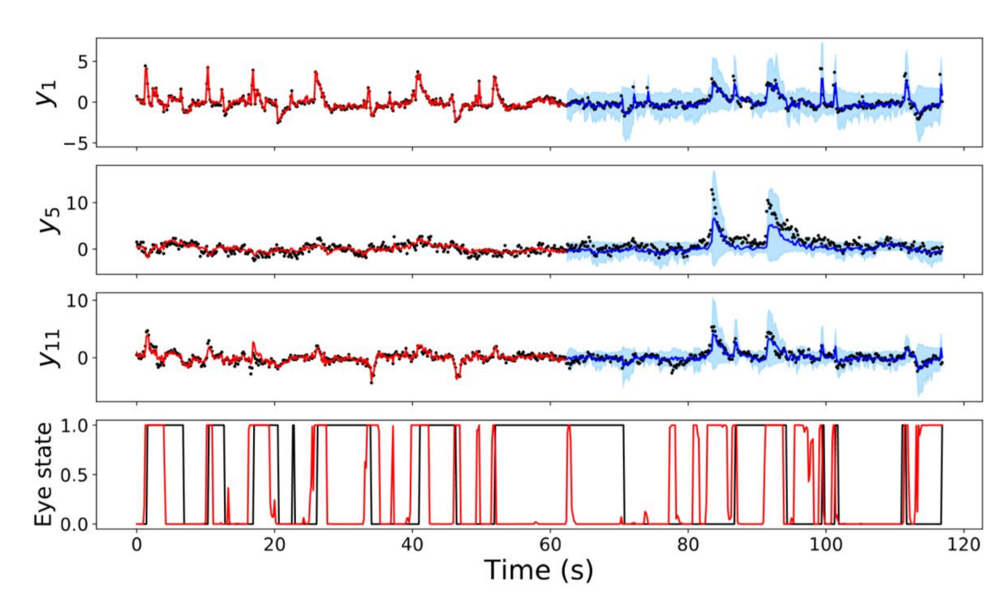
Funsor expressions are closed under these approximation rewrites.

Example: detecting EEG changepoints

We use a moment matching interpreter to fit a switching linear dynamical system:

```
@interpretation(moment_matching_interpretation)
def marginal_log_prob(data, s_trans, x_trans, y_dist):
  ...
  for t, y in enumerate(data):
    ss[t] = Variable(f"s_{t}", bint(2))
    xs[t] = Variable(f"x_{t}", reals(5))
    ...
    log_prob += fdist.Categorical(
      s_trans(s=ss[t - 1]), value=ss[t])
    log_prob += x_trans(s=ss[t], x=xs[t - 1], y=xs[t])

    log_prob = log_prob.reduce(ops.logaddexp,
      {ss[t - 2].name, xs[t - 2].name})

    log_prob += y_dist(s=s_vars[t], x=x_vars[t], y=y)

  for t in range(2):
    log_prob = log_prob.reduce(ops.logaddexp,
      {s_vars[T - 2 + t].name, x_vars[T - 2 + t].name})

  return log_prob
```





Extending the language: parallel-scan over sequential structure

Many common sum-product expressions have linear chain structures.
We define a generic operation on atomic funsor terms for collapsing this structure:

For Tensor terms, the MarkovProduct op corresponds to chain matrix multiplication:

$$\prod_{t/(i,j)} f = f[t=0] \bullet f[t=1] \bullet \cdots \bullet f[t=T-1]$$

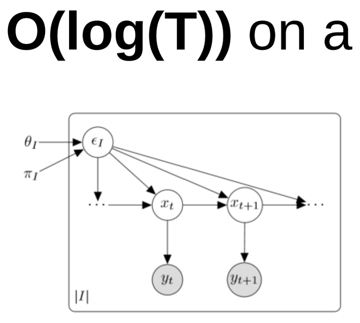
i.e. the binary operation is a GEMM:

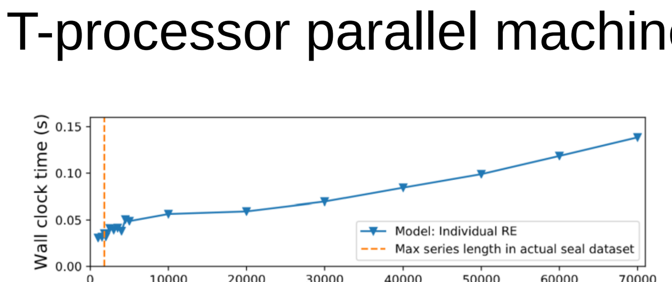
$$f \bullet g = \sum_k f[j=k] \times g[i=k]$$

Our parallel-scan algorithm computes this in **O(log(T))** on a T-processor parallel machine:

Algorithm 1 MarkovProduct

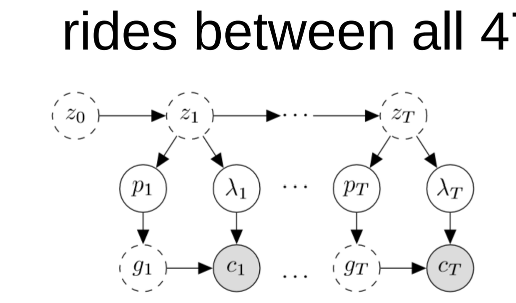
input a funsor f, a time variable $t \in \mathrm{fv}(f)$,
 a step mapping $s \subseteq \mathrm{fv}(f) \times \mathrm{fv}(f)$.
output the Markov product funsor $\prod_{t/s} f$.
Create substitutions with fresh names (barred):
$s_e \leftarrow \{(y, \bar{x}) \mid (x, y) \in s\}$ to rename even factors, and
$s_o \leftarrow \{(x, \bar{x}) \mid (x, y) \in s\}$ to rename odd factors.
Let $v \leftarrow \{\bar{x} \mid (x, y) \in s\}$ be variables to marginalize.
Let $T \leftarrow |\Gamma_f[t]|$ be the length of the time axis.
while $T > 1$ **do**
 Split f into even and odd parts of equal length:
 $f_e \leftarrow f[s_e, t=(0, 2, 4, 6, ..., 2\lfloor T/2 \rfloor - 2)]$
 $f_o \leftarrow f[s_o, t=(1, 3, 5, 7, ..., 2\lfloor T/2 \rfloor - 1)]$
 Perform parallel sum-product contraction:
 $f' \leftarrow \sum_v f_e \times f_o$
 if T *is even* **then** $f \leftarrow f'$
 else $f \leftarrow \mathrm{concat}_t(f', f[t=T-1])$;
 $T \leftarrow \lceil T/2 \rceil$
return $f[t=0]$

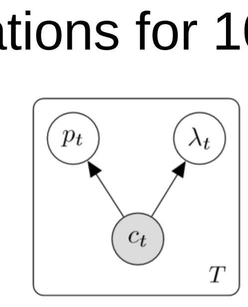


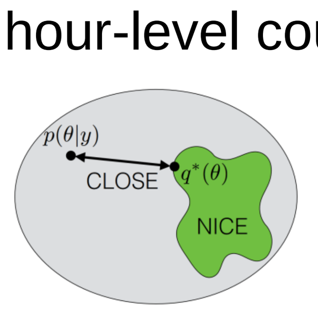


Example: forecasting BART ridership

We do collapsed variational inference in a neural Kalman filter to model rides between all 47 BART stations for 10 years of hour-level counts:








```
def model(features, counts):
  gr_t = funsor.Variable(
    "gr_t", ...)["time"]
  init, trans, obs, rate = ...
  ...
  prior = MarkovProduct(
    ops.logaddexp, ops.add,
    trans + obs(gate_rate=gr_t),
    "time", {"state": "state(time=1)"})

  llk = fdist.Poisson(rate["origin", "destin"])
  return prior + llk(value=counts)
```

```
def guide(features, counts):
  loc, sc = ...
  diag_normal = fdist.Normal(
    loc, sc, value=“gr_t”)
  return diag_normal
```

```
def elbo_loss(features, counts):
  q = guide(features, counts)

  with interpretation(lazy):
    p = model(features, counts)
    pq = p - q

  with interpretation(monte_carlo):
    elbo = funsor.Integrate(q, pq)

  return elbo
```