**Funson: Functional Tensors for Probabilistic Programming**


**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 between fully static and fully black-box integration.

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

We extend lazy tensor expressions to include dimensions of size 'real' and implement semisymbolic integration

**Operational semantics:** designing for performance and ease of implementation

The term language has an obvious default operational semantics backed by high-performance tensor libraries like JAX and PyTorch. This not only gives us differentiable, hardware-accelerated kernels, but provides multiple tracing JIT compilers that compile away all runtime pattern-matching overhead.

**Approximation and transformation via (re-)interpretation**

Most integrals defined by Funson terms cannot be computed directly. We rewrite lazy expressions by evaluating them with many different interpreters.

Some rules trigger PyTorch ops: eager.register, eager.add, eager.log, eager.logaddexp, eager.pexp, eager.prod, eager.reduce, eager.all

Some triggers further rewrites:

Some rewrite subexpressions into equivalent versions, monte_carlo.rewrite Tensor and Gaussian to Delta:

We use a moment matching interpreter to approximate posterior expectations:

Funsor provides a generic framework for automatic integration over array-structured data.

**Operational semantics:** term rewriting with a hierarchy of tagless final interpreters

Sometimes expressions are expensive to evaluate directly or need to be simplified for pattern-matching with existing rewrite rules.

We rewrite these expressions with interpreters that preserve their exact semantics.

(a) Gaussian terms are unnormalized block-structured multivariate Gaussian densities.
(b) Gaussian terms with continuous dimensions are replaced by a term representing the multivariate Gaussian
(c) Funsor and PyTorch expressions are evaluated using the same interpreter

**Example:** detecting EEG changepoints

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

We can apply these interpreters to evaluate sum-product expressions with any commutative semiring, including quantifying uncertainty, learning structured generative models, producing interpretable explanations of data, and learning from weak or missing labels.

**Operational semantics:** closure under approximation

Some expressions have no equivalent form under exact semantics. The term language was carefully chosen to be closed under popular approximations which are interpreters that preserve types but not semantics. This allows us to evaluate all expressions semi-numerically (rather than symbolically).