NumPyro: Using Composable Effects for Flexible and Accelerated Probabilistic Programming

https://num.pyro.ai

Du Phan (UIUC)

Neeraj Pradhan (Facebook)

Martin Jankowiak (Broad Institute)

Probabilistic Modeling with JAX

NumPyro is a library for probabilistic inference built on JAX, that has the same interface for model specification and inference as Pyro.

JAX is a high-level tracing library for program transformations of Python and NumPy functions. e.g. automatic differentiation (`grad`), JIT compilation (`jit`), vectorization (`vmap`), and parallelization (`pmap`). Inference subroutines in NumPyro use effect handlers to inspect and modify program behavior and freely compose with JAX transformations resulting in significant speedup via **parallelization** and **JIT compilation**.

Support for Pyro Primitives

```
# declare a trainable param
p = numpyro.param("p", np.ones(10),
                  constraint=positive)

# sample a random value
x = numpyro.sample("x", Normal(0, p))

# declare a batch dimension
with numpyro.plate("data", y.shape[0]):

    # observe a random variable
    numpyro.sample("y", Normal(x, 1), obs=y)
```

Automatic Enumeration of Discrete Latent Variables

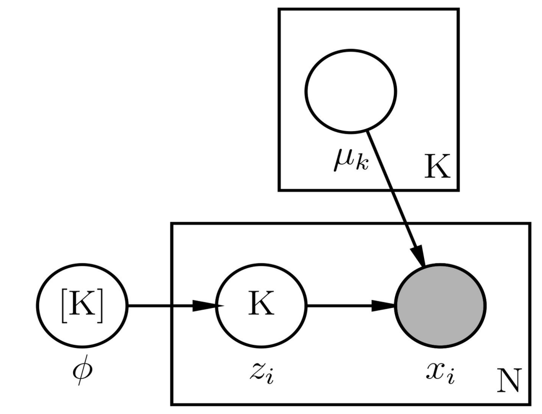
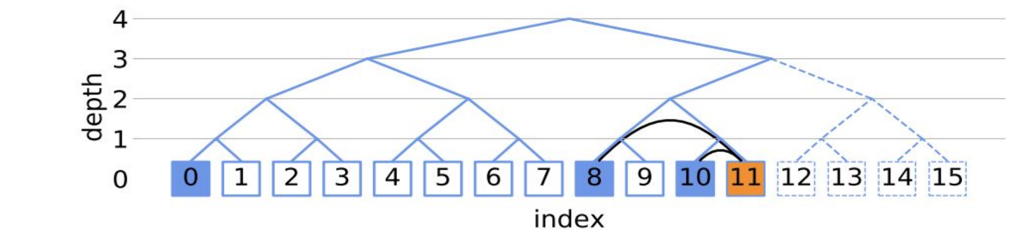


Plate notation of Gaussian Mixture Model

```
def gmm(data, K):
    phi = sample("phi", Dirichlet(np.ones(K)))
    with plate("K", K, dim=-1):
        mu = sample("mu", Normal(np.arange(K), 1))
    with plate("N", len(data), dim=-1):
        z = sample("z", Categorical(phi))
        sample("obs", Normal(mu[z], 1), obs=data)
```

Effect handlers allow to modify the behavior of the program, hence enable more advanced inference mechanism such as enumeration to marginalize out the discrete latent variable "z". In particular, effect handlers allow us to run the program in two modes: one in which discrete latent variables are sampled and one in which they are enumerated. The first mode can be used to inspect the model structure and the second mode is used to compute the joint probability of the model.

Speeding up NUTS via JIT Compilation — Iterative NUTS



A graphical representation of how binary trees are constructed in IterativeBuildTree. The orange node is the leaf generated at the current step. Blue nodes are the leaves stored in memory for the purpose of checking the U-Turn condition. White nodes are past leaves that have been removed from memory.

Algorithm 2 IterativeBuildTree

Input initial node z, tree depth d

Initialize storage $S[0], S[1], ..., S[d-1]$

for $n \leftarrow 0$ **to** $2^d - 1$ **do**

  $z \leftarrow \text{Leapfrog}(z)$

  if n *is even* **then**

    $i \leftarrow \text{BitCount}(n)$

    $S[i] \leftarrow z$

  else

    // gets the number of candidate nodes

    $l \leftarrow \text{TrailingBit}(n)$

    $i_{max} \leftarrow \text{BitCount}(n-1)$

    $i_{min} \leftarrow i_{max} - l + 1$

    for $k \leftarrow i_{max}$ **to** i_{min} **do**

      $turning \leftarrow \text{IsUTurn}(S[k], z)$

      if $turning$ **then**

        return $\text{Tree}(S[0], z, True)$

return $\text{Tree}(S[0], z, False)$

Memory Efficiency: Store only even numbered nodes z_k at indices given by BitCount(k). Requires O(log N) memory.

Effect Handlers

Effect handlers provide a way to inject effectful computation into primitive statements in a probabilistic program, e.g. recording the random choices made in an execution trace.

This lets us:

- Expose a unified modeling and inference interface that is largely the same as Pyro.
- Speed up critical subroutines via parallelization and JIT compilation, since these effects can be freely composed with JAX transformations.
- Enable parallel enumeration of discrete latent variables, reparameterization such as loc-scale decentering and neural transport for HMC.

Some basic examples of *effect handlers*:

`seed`

- Seeds `fn` with a `PRNGKey`. Every call to sample inside fn results in splitting of PRNGKey to generate a fresh seed for subsequent calls.
- `seed(fn, rng)(...)`

`trace`

- Records the input, output, and function calls inside of sample, param statements in `fn`.
- `trace(fn).get_trace(...)`

`condition`

- Conditions unobserved sample sites in `fn` to values in `data`.
- `condition(fn, data)(...)`

It is easy to write fast vectorized inference utilities by combining effect handlers like `seed`, `condition` and `trace` with JAX transformations like `vmap` and `jit`.

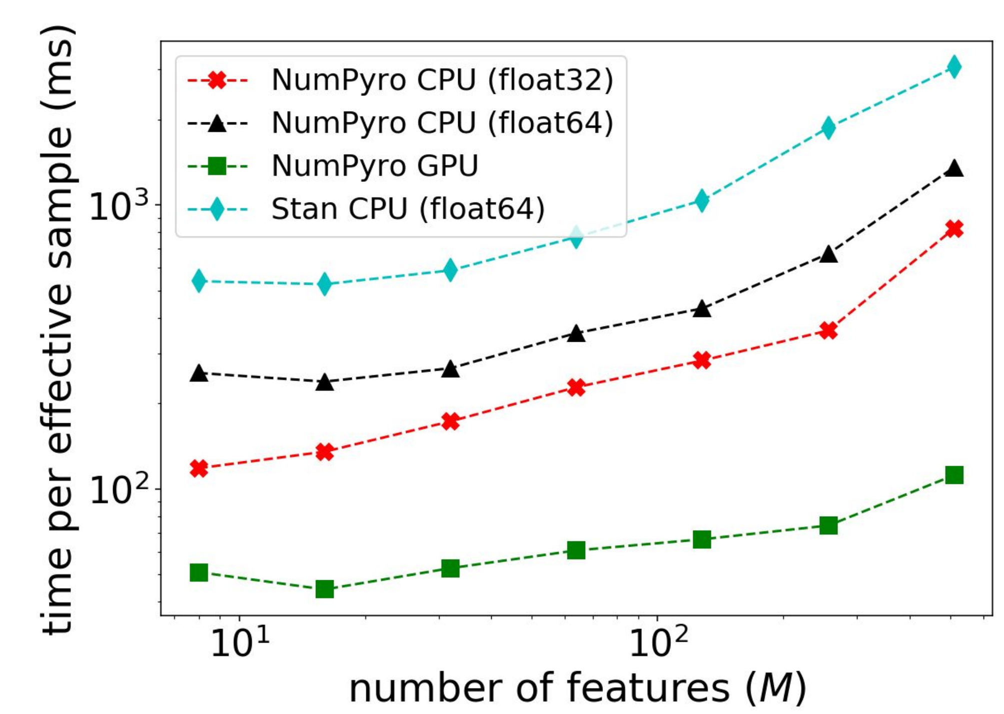
```
def logistic_regression(x, y=None):
  ndims = np.shape(x)[-1]
  m = numpyro.sample('m', Normal(0, 1).expand([ndims]))
  b = numpyro.sample('b', Normal(0, 1))
  return numpyro.sample('y', Bernoulli(logits=x @ m + b),
                        obs=y)

# Run inference to generate samples from the posterior
kernel = NUTS(model=logistic_regression)
mcmc = MCMC(kernel, num_warmup, num_samples)
mcmc.run(random.PRNGKey(1), x, y=y)
samples = mcmc.get_samples()

def predict_fn(rng_key, param, *args):
  conditioned_model = condition(logistic_regression, param)
  return seed(conditioned_model, rng_key)(*args)

# Generate batch of PRNGKeys
rngs_sim = random.split(random.PRNGKey(2), num_samples)
# Vectorized prediction using vmap
posterior_predictive = vmap(lambda rng_key, param:
    predict_fn(rng_key, param, x))(rng_keys_pred, samples)
```

Fast Inference for Both Small and Large Dataset Regimes

Time taken per leapfrog step (in ms.)

Framework	HMM[6]	COVTYPE
Stan (64-bit CPU)	0.53	135.94
Pyro (32-bit CPU)	30.51	32.76
Pyro (GPU)	-	3.36
NumPyro (32-bit CPU)	**0.09**	30.11
NumPyro (64-bit CPU)	0.15	71.18
NumPyro (GPU)	-	**1.46**

[6] Average effective sample size with 1000 warmup steps and 1000 samples for each run in Stan, NumPyro (32 bit), and NumPyro (64 bit) are 652, 556, and 778 respectively.



Time taken per effective sample (in ms.) for different frameworks on the Sparse Kernel Interaction Model (SKIM) example using NUTS, as the number of features (M) is varied.

Conclusion

- NumPyro is a library for doing probabilistic inference. It is batteries included with modules for distributions, bijective transforms, and effect handlers.
- NumPyro uses JAX transformations under the hood for hardware acceleration, automatic differentiation, and vectorization.
- NumPyro's effect handlers are composable with JAX's transformations. This composability allows us to
  - offer the same modeling language as Pyro with features such as automatic enumeration of discrete latent variables.
  - leverage JAX transformations to parallelize and JIT compile static inference subroutines for significant speed ups.