tfp.mcmc: Modern Markov Chain Monte Carlo Tools Built for Modern Hardware

Junpeng Lao, Christopher Suter, Ian Langmore, Cyril Chimisov, Ashish Saxena, Pavel Sountsov, Dave Moore, Rif A. Saurous, Matthew D. Hoffman, and Joshua V. Dillon

Google Research

Key takeaways

What: `tfp.mcmc` is a highly flexible and modular framework for MCMC research and Bayesian inference, focused on performance, and built on top of TensorFlow and Jax.

How:

- Pervasive Data Parallelism (using "batch semantics" that leverage "single instruction, multiple data" (SIMD) instruction sets ("data parallelism")
- Requires only a simple Python callable that maps `inputs -> log_prob`, where `inputs` is a nested Python structure
- `TransitionKernels` and `drivers` that can be nested together to create new MCMC routines

Where: tfp.mcmc and tfp.experimental.mcmc

TransitionKernel

```
class TransitionKernel:
 @abc.abstractmethod
 def one_step(self, current_state, previous_kernel_results, seed=None):
   """Takes one step of the TransitionKernel."""
   ...
 def bootstrap_results(self, init_state):
   """Returns an object with the same type as returned by `one_step(...)[1]`."""
   ...
 @abc.abstractproperty
 def is_calibrated(self):
   """Returns `True` if Markov chain converges to specified distribution."""
   ...
```

TransitionKernel's are composable

```
randomwalk_mh = tfp.mcmc.MetropolisHastings(
   inner_kernel=tfp.mcmc.UncalibratedRandomWalk(
       target_log_prob_fn=target_log_prob_fn,
       new_state_fn=new_state_fn))

hmc = tfp.mcmc.MetropolisHastings(
   inner_kernel=tfp.mcmc.UncalibratedHamiltonianMonteCarlo(
       target_log_prob_fn=target_log_prob_fn,
       step_size=step_size))

hmc_unbounded_with_tuning = tfp.mcmc.DualAveragingStepSizeAdaptation(
    tfp.mcmc.TransformedTransitionKernel(inner_kernel=hmc, bijector=bijector),
    target_accept_prob=.8, num_adaptation_steps=burnin)

# Another design pattern we use is to have a make_kernel_fn that generates a TK
def make_kernel_fn(log_prob_fn):
 return tfp.mcmc.HamiltonianMonteCarlo(
     log_prob_fn, step_size=step_size, num_leapfrog_steps=10)
remc = tfp.mcmc.ReplicaExchangeMC(
   target_log_prob_fn=target_log_prob_fn,
   inverse_temperatures=inverse_temperatures,
   make_kernel_fn=make_kernel_fn)
```

drivers

```
def driver(kernel, initial_state):
 [] = results
 side_results = kernel.bootstrap_results(initial_state)
 for _ in range(num_samples):
   x, side_results = kernel.one_step(results[-1], side_results)
   results += [x]
 return results
results = driver(SomeKernel(target_log_prob_callable), x0)
```

driver examples

```
def trace_fn(state, adaptive_pkr):
 """`adaptive_pkr` is the previous kernel result."""
 transformed_pkr = adaptive_pkr.inner_results
 metropolis_pkr = transformed_pkr.inner_results
 return metropolis_pkr.is_accepted
# Draw 500 samples, and trace the MH acceptance outcomes.
samples, is_accepted = tfp.mcmc.sample_chain(
   current_state=init_state,
   kernel=hmc_unbounded_with_tuning,
   num_burnin_steps=300, num_results=500,
   trace_fn=trace_fn)

cov_reducer = tfp.experimental.mcmc.CovarianceReducer()
covariance_estimate, _, _ = tfp.experimental.mcmc.sample_fold(
   current_state=init_state,
   kernel=hmc_unbounded_with_tuning,
   num_burnin_steps=300, num_results=500,
   trace_fn=trace_fn,
   reducers=cov_reducer,
)

smc_result = sample_sequential_monte_carlo(
   prior_log_prob_fn,
   likelihood_log_prob_fn,
   current_state,
   make_kernel_fn=make_rwmh_kernel_fn)
```

Highlights of some recent new features

```
# New TransitionKernels
tfp.experimental.mcmc.GradientBasedTrajectoryLengthAdaptation
tfp.experimental.mcmc.PreconditionedHamiltonianMonteCarlo
tfp.experimental.mcmc.SampleDiscardingKernel

# New sample drivers
tfp.experimental.mcmc.sample_sequential_monte_carlo
tfp.experimental.mcmc.sample_fold

# `Reducer` that accumulates trace results at each sample.
tfp.experimental.mcmc.ProgressBarReducer
tfp.experimental.mcmc.ExpectationsReducer
tfp.experimental.mcmc.CovarianceReducer
tfp.experimental.mcmc.PotentialScaleReductionReducer
tfp.experimental.mcmc.TracingReducer
```

Discussion

Advantages and challenges of pervasive data parallelism

Q: Why is the pervasive data parallelism advantageous? Can't we just use vectorizing function like `tf.vectorized_map` or `jax.vmap` and wrap the TK into a SIMD function?

A: Pervasive data parallelism opens new opportunities to directly manipulate across "batches", even *during* one MCMC step. For example, we can flexibly implement population-wise MCMC methods, or coupling MCMC methods.

There are also significant challenges, for example, in the implementation of the NUTS sampler.

Challenges of being modular

Onion-like nesting `TransitionKernels` are powerful, but also create challenges when we try to access some properties in one of the layer of the `kernel_results`. We have made some progress to make this process easier with tfp.experimental.unnest

Contact

https://www.tensorflow.org/probability/

Reach out to us on our Google group if you have any questions: **`tfprobability@tensorflow.org`**