tf.contrib.bayesflow.hmc.chain

```
chain(
    n_iterations,
    step_size,
    n_leapfrog_steps,
    initial_x,
    target_log_prob_fn,
    event_dims=(),
    name=None
)
```

Defined in `tensorflow/contrib/bayesflow/python/ops/hmc_impl.py`.

Runs multiple iterations of one or more Hamiltonian Monte Carlo chains.

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) algorithm that takes a series of gradient-informed steps to produce a Metropolis proposal. This function samples from an HMC Markov chain whose initial state is `initial_x` and whose stationary distribution has log-density `target_log_prob_fn()`.

This function can update multiple chains in parallel. It assumes that all dimensions of `initial_x` not specified in `event_dims` are independent, and should therefore be updated independently. The output of `target_log_prob_fn()` should sum log-probabilities across all event dimensions. Slices along dimensions not in `event_dims` may have different target distributions; this is up to `target_log_prob_fn()`.

This function basically just wraps `hmc.kernel()` in a tf.scan() loop.

Args:

- `n_iterations`: Integer number of Markov chain updates to run.
- `step_size`: Scalar step size or array of step sizes for the leapfrog integrator. Broadcasts to the shape of `initial_x`. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it's often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.
- `n_leapfrog_steps`: Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to step_size * n_leapfrog_steps.
- `initial_x`: Tensor of initial state(s) of the Markov chain(s).
- `target_log_prob_fn`: Python callable which takes an argument like `initial_x` and returns its (possibly unnormalized) log-density under the target distribution.
- `event_dims`: List of dimensions that should not be treated as independent. This allows for multiple chains to be run independently in parallel. Default is (), i.e., all dimensions are independent.
- `name`: Python `str` name prefixed to Ops created by this function.

Returns:

- `acceptance_probs`: Tensor with the acceptance probabilities for each iteration. Has shape matching `target_log_prob_fn(initial_x)`.
- `chain_states`: Tensor with the state of the Markov chain at each iteration. Has shape `[n_iterations,`

`initial_x.shape[0],...,initial_x.shape[-1]` .

Examples:

```
# Sampling from a standard normal (note `log_joint()` is unnormalized):
def log_joint(x):
  return tf.reduce_sum(-0.5 * tf.square(x))
chain, acceptance_probs = hmc.chain(1000, 0.5, 2, tf.zeros(10), log_joint,
                                    event_dims=[0])
# Discard first half of chain as warmup/burn-in
warmed_up = chain[500:]
mean_est = tf.reduce_mean(warmed_up, 0)
var_est = tf.reduce_mean(tf.square(warmed_up), 0) - tf.square(mean_est)
```

```
# Sampling from a diagonal-variance Gaussian:
variances = tf.linspace(1., 3., 10)
def log_joint(x):
  return tf.reduce_sum(-0.5 / variances * tf.square(x))
chain, acceptance_probs = hmc.chain(1000, 0.5, 2, tf.zeros(10), log_joint,
                                    event_dims=[0])
# Discard first half of chain as warmup/burn-in
warmed_up = chain[500:]
mean_est = tf.reduce_mean(warmed_up, 0)
var_est = tf.reduce_mean(tf.square(warmed_up), 0) - tf.square(mean_est)
```

```
# Sampling from factor-analysis posteriors with known factors W:
# mu[i, j] ~ Normal(0, 1)
# x[i] ~ Normal(matmul(mu[i], W), I)
def log_joint(mu, x, W):
  prior = -0.5 * tf.reduce_sum(tf.square(mu), 1)
  x_mean = tf.matmul(mu, W)
  likelihood = -0.5 * tf.reduce_sum(tf.square(x - x_mean), 1)
  return prior + likelihood
chain, acceptance_probs = hmc.chain(1000, 0.1, 2,
                                    tf.zeros([x.shape[0], W.shape[0]]),
                                    lambda mu: log_joint(mu, x, W),
                                    event_dims=[1])
# Discard first half of chain as warmup/burn-in
warmed_up = chain[500:]
mean_est = tf.reduce_mean(warmed_up, 0)
var_est = tf.reduce_mean(tf.square(warmed_up), 0) - tf.square(mean_est)
```

```
# Sampling from the posterior of a Bayesian regression model.:

# Run 100 chains in parallel, each with a different initialization.
initial_beta = tf.random_normal([100, x.shape[1]])
chain, acceptance_probs = hmc.chain(1000, 0.1, 10, initial_beta,
                                    log_joint_partial, event_dims=[1])
# Discard first halves of chains as warmup/burn-in
warmed_up = chain[500:]
# Averaging across samples within a chain and across chains
mean_est = tf.reduce_mean(warmed_up, [0, 1])
var_est = tf.reduce_mean(tf.square(warmed_up), [0, 1]) - tf.square(mean_est)
```

Last updated November 2, 2017.

Stay Connected

Blog

GitHub

Twitter

Support

Issue Tracker

Release Notes

Stack Overflow

English

Terms | **Privacy**