tf.contrib.bayesflow.hmc.ais_chain

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

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

Runs annealed importance sampling (AIS) to estimate normalizing constants.

This routine uses Hamiltonian Monte Carlo to sample from a series of distributions that slowly interpolates between an initial "proposal" distribution

`exp(proposal_log_prob_fn(x) - proposal_log_normalizer)`

and the target distribution

`exp(target_log_prob_fn(x) - target_log_normalizer)`,

accumulating importance weights along the way. The product of these importance weights gives an unbiased estimate of the ratio of the normalizing constants of the initial distribution and the target distribution:

E[exp(w)] = exp(target_log_normalizer - proposal_log_normalizer).

Args:

- `n_iterations`: Integer number of Markov chain updates to run. More iterations means more expense, but smoother annealing between q and p, which in turn means exponentially lower variance for the normalizing constant estimator.
- `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). Must be a sample from q, or results will be incorrect.
- `target_log_prob_fn`: Python callable which takes an argument like `initial_x` and returns its (possibly unnormalized) log-density under the target distribution.
- `proposal_log_prob_fn`: Python callable that returns the log density of the initial 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:

- `ais_weights` : Tensor with the estimated weight(s). Has shape matching **`target_log_prob_fn(initial_x)`** .
- `chain_states` : Tensor with the state(s) of the Markov chain(s) the final iteration. Has shape matching **`initial_x`** .
- `acceptance_probs` : Tensor with the acceptance probabilities for the final iteration. Has shape matching **`target_log_prob_fn(initial_x)`** .

Examples:

```
# Estimating the normalizing constant of a log-gamma distribution:
def proposal_log_prob(x):
  # Standard normal log-probability. This is properly normalized.
  return tf.reduce_sum(-0.5 * tf.square(x) - 0.5 * np.log(2 * np.pi), 1)
def target_log_prob(x):
  # Unnormalized log-gamma(2, 3) distribution.
  # True normalizer is (lgamma(2) - 2 * log(3)) * x.shape[1]
  return tf.reduce_sum(2. * x - 3. * tf.exp(x), 1)
# Run 100 AIS chains in parallel
initial_x = tf.random_normal([100, 20])
w, _, _ = hmc.ais_chain(1000, 0.2, 2, initial_x, target_log_prob,
                        proposal_log_prob, event_dims=[1])
log_normalizer_estimate = tf.reduce_logsumexp(w) - np.log(100)
```

```
# Estimating the marginal likelihood of a Bayesian regression model:
base_measure = -0.5 * np.log(2 * np.pi)
def proposal_log_prob(x):
  # Standard normal log-probability. This is properly normalized.
  return tf.reduce_sum(-0.5 * tf.square(x) + base_measure, 1)
def regression_log_joint(beta, x, y):
  # This function returns a vector whose ith element is log p(beta[i], y | x).
  # Each row of beta corresponds to the state of an independent Markov chain.
  log_prior = tf.reduce_sum(-0.5 * tf.square(beta) + base_measure, 1)
  means = tf.matmul(beta, x, transpose_b=True)
  log_likelihood = tf.reduce_sum(-0.5 * tf.square(y - means) +
                                 base_measure, 1)
  return log_prior + log_likelihood
def log_joint_partial(beta):
  return regression_log_joint(beta, x, y)
# Run 100 AIS chains in parallel
initial_beta = tf.random_normal([100, x.shape[1]])
w, beta_samples, _ = hmc.ais_chain(1000, 0.1, 2, initial_beta,
                                   log_joint_partial, proposal_log_prob,
                                   event_dims=[1])
log_normalizer_estimate = tf.reduce_logsumexp(w) - np.log(100)
```