tf.contrib.bayesflow.metropolis_hastings.evolve

```
evolve(
    initial_sample,
    initial_log_density,
    initial_log_accept_ratio,
    log_unnormalized_prob_fn,
    proposal_fn,
    n_steps=1,
    seed=None,
    name=None
)
```

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

Performs `n_steps` of the Metropolis-Hastings update.

Given a probability density function, `f(x)` and a proposal scheme which generates new points from old, this `Op` returns a tensor which may be used to generate approximate samples from the target distribution using the Metropolis-Hastings algorithm. These samples are from a Markov chain whose equilibrium distribution matches the target distribution.

The probability distribution may have an unknown normalization constan. We parameterize the probability density as follows:

```
  f(x) = exp(L(x) + constant)
```

Here `L(x)` is any continuous function with an (possibly unknown but finite) upper bound, i.e. there exists a number beta such that `L(x)< beta < infinity` for all x. The constant is the normalization needed to make `f(x)` a probability density (as opposed to just a finite measure).

Although `initial_sample` can be arbitrary, a poor choice may result in a slow-to-mix chain. In many cases the best choice is the one that maximizes the target density, i.e., choose `initial_sample` such that `f(initial_sample) >= f(x)` for all `x`.

If the support of the distribution is a strict subset of R^n (but of non zero measure), then the unnormalized log-density `L(x)` should return `-infinity` outside the support domain. This effectively forces the sampler to only explore points in the regions of finite support.

Usage: This function is meant to be wrapped up with some of the common proposal schemes (e.g. random walk, Langevin diffusion etc) to produce a more user friendly interface. However, it may also be used to create bespoke samplers.

The following example, demonstrates the use to generate a 1000 uniform random walk Metropolis samplers run in parallel for the normal target distribution.

```
n = 3  # dimension of the problem

# Generate 1000 initial values randomly. Each of these would be an
# independent starting point for a Markov chain.
state = tf.get_variable(
    'state',initializer=tf.random_normal([1000, n], mean=3.0,
                                         dtype=tf.float64, seed=42))

# Computes the log(p(x)) for the unit normal density and ignores the
# normalization constant.
def log_density(x):
  return  - tf.reduce_sum(x * x, reduction_indices=-1) / 2.0

# Initial log-density value
state_log_density = tf.get_variable(
    'state_log_density', initializer=log_density(state.initialized_value()))

# A variable to store the log_acceptance_ratio:
log_acceptance_ratio = tf.get_variable(
    'log_acceptance_ratio', initializer=tf.zeros([1000], dtype=tf.float64))

# Generates random proposals by moving each coordinate uniformly and
# independently in a box of size 2 centered around the current value.
# Returns the new point and also the log of the Hastings ratio (the
# ratio of the probability of going from the proposal to origin and the
# probability of the reverse transition). When this ratio is 1, the value
# may be omitted and replaced by None.
def random_proposal(x):
  return (x + tf.random_uniform(tf.shape(x), minval=-1, maxval=1,
                                dtype=x.dtype, seed=12)), None

#  Create the op to propagate the chain for 100 steps.
stepper = mh.evolve(
    state, state_log_density, log_acceptance_ratio,
    log_density, random_proposal, n_steps=100, seed=123)
init = tf.initialize_all_variables()
with tf.Session() as sess:
  sess.run(init)
  # Run the chains for a total of 1000 steps and print out the mean across
  # the chains every 100 iterations.
  for n_iter in range(10):
    # Executing the stepper advances the chain to the next state.
    sess.run(stepper)
    # Print out the current value of the mean(sample) for every dimension.
    print(np.mean(sess.run(state), 0))
  # Estimated covariance matrix
  samples = sess.run(state)
  print('')
  print(np.cov(samples, rowvar=False))
```

Args:

- `initial_sample`: A float-like `tf.Variable` of any shape that can be consumed by the `log_unnormalized_prob_fn` and `proposal_fn` callables.
- `initial_log_density`: Float-like `tf.Variable` with `dtype` and shape equivalent to `log_unnormalized_prob_fn(initial_sample)`, i.e., matching the result of `log_unnormalized_prob_fn` invoked at `current_state`.
- `initial_log_accept_ratio`: A `tf.Variable` with `dtype` and shape matching `initial_log_density`. Stands for the log of Metropolis-Hastings acceptance ratio after propagating the chain for `n_steps`.
- `log_unnormalized_prob_fn`: A Python callable evaluated at `current_state` and returning a float-like `Tensor` of log

target-density up to a normalizing constant. In other words, `log_unnormalized_prob_fn(x) = log(g(x))`, where `target_density = g(x)/Z` for some constant `A`. The shape of the input tensor is the same as the shape of the `current_state`. The shape of the output tensor is either (a). Same as the input shape if the density being sampled is one dimensional, or (b). If the density is defined for `events` of shape `event_shape = [E1, E2, ... Ee]`, then the input tensor should be of shape `batch_shape + event_shape`, here `batch_shape = [B1, ..., Bb]` and the result must be of shape [B1, ..., Bb]. For example, if the distribution that is being sampled is a 10 dimensional normal, then the input tensor may be of shape [100, 10] or [30, 20, 10]. The last dimension will then be 'consumed' by `log_unnormalized_prob_fn` and it should return tensors of shape [100] and [30, 20] respectively.

- `proposal_fn`: A callable accepting a real valued `Tensor` of current sample points and returning a tuple of two `Tensors`. The first element of the pair should be a `Tensor` containing the proposal state and should have the same shape as the input `Tensor`. The second element of the pair gives the log of the ratio of the probability of transitioning from the proposal points to the input points and the probability of transitioning from the input points to the proposal points. If the proposal is symmetric, i.e. Probability(Proposal -> Current) = Probability(Current -> Proposal) the second value should be set to None instead of explicitly supplying a tensor of zeros. In addition to being convenient, this also leads to a more efficient graph.
- `n_steps`: A positive `int` or a scalar `int32` tensor. Sets the number of iterations of the chain.
- `seed`: `int` or None. The random seed for this `Op`. If `None`, no seed is applied.
- `name`: A string that sets the name for this `Op`.

Returns:

- `forward_step`: an `Op` to step the Markov chain forward for `n_steps`.