tf.contrib.bayesflow.hmc.kernel

```
kernel(
    step_size,
    n_leapfrog_steps,
    x,
    target_log_prob_fn,
    event_dims=(),
    x_log_prob=None,
    x_grad=None,
    name=None
)
```

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

Runs one iteration of Hamiltonian Monte Carlo.

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 applies one step of HMC to randomly update the variable `x`.

This function can update multiple chains in parallel. It assumes that all dimensions of `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; for example, if `event_dims == (1,)`, then `x[0, :]` could have a different target distribution from x[1, :]. This is up to `target_log_prob_fn()`.

Args:

- `step_size`: Scalar step size or array of step sizes for the leapfrog integrator. Broadcasts to the shape of `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.
- `x`: Tensor containing the value(s) of the random variable(s) to update.
- `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. x_log_prob (optional): Tensor containing the cached output of a previous call to `target_log_prob_fn()` evaluated at `x` (such as that provided by a previous call to `kernel()`). Providing `x_log_prob` and `x_grad` saves one gradient computation per call to `kernel()`. x_grad (optional): Tensor containing the cached gradient of `target_log_prob_fn()` evaluated at `x` (such as that provided by a previous call to `kernel()`). Providing `x_log_prob` and `x_grad` saves one gradient computation per call to `kernel()`.
- `name`: Python `str` name prefixed to Ops created by this function.

Returns:

- `updated_x`: The updated variable(s) x. Has shape matching `initial_x`.

- `acceptance_probs`: Tensor with the acceptance probabilities for the final iteration. This is useful for diagnosing step size problems etc. Has shape matching **`target_log_prob_fn(initial_x)`**.
- `new_log_prob`: The value of **`target_log_prob_fn()`** evaluated at **`updated_x`**.
- `new_grad`: The value of the gradient of **`target_log_prob_fn()`** evaluated at **`updated_x`**.

Examples:

```
# Tuning acceptance rates:
target_accept_rate = 0.631
def target_log_prob(x):
  # Standard normal
  return tf.reduce_sum(-0.5 * tf.square(x))
initial_x = tf.zeros([10])
initial_log_prob = target_log_prob(initial_x)
initial_grad = tf.gradients(initial_log_prob, initial_x)[0]
# Algorithm state
x = tf.Variable(initial_x, name='x')
step_size = tf.Variable(1., name='step_size')
last_log_prob = tf.Variable(initial_log_prob, name='last_log_prob')
last_grad = tf.Variable(initial_grad, name='last_grad')
# Compute updates
new_x, acceptance_prob, log_prob, grad = hmc.kernel(step_size, 3, x,
                                                    target_log_prob,
                                                    event_dims=[0],
                                                    x_log_prob=last_log_prob)
x_update = tf.assign(x, new_x)
log_prob_update = tf.assign(last_log_prob, log_prob)
grad_update = tf.assign(last_grad, grad)
step_size_update = tf.assign(step_size,
                             tf.where(acceptance_prob > target_accept_rate,
                                      step_size * 1.01, step_size / 1.01))
adaptive_updates = [x_update, log_prob_update, grad_update, step_size_update]
sampling_updates = [x_update, log_prob_update, grad_update]

sess = tf.Session()
sess.run(tf.global_variables_initializer())
# Warm up the sampler and adapt the step size
for i in xrange(500):
  sess.run(adaptive_updates)
# Collect samples without adapting step size
samples = np.zeros([500, 10])
for i in xrange(500):
  x_val, _ = sess.run([new_x, sampling_updates])
  samples[i] = x_val
```

```
# Empirical-Bayes estimation of a hyperparameter by MCMC-EM:

# Problem setup
N = 150
D = 10
x = np.random.randn(N, D).astype(np.float32)
true_sigma = 0.5
true_beta = true_sigma * np.random.randn(D).astype(np.float32)
y = x.dot(true_beta) + np.random.randn(N).astype(np.float32)

def log_prior(beta, log_sigma):
  return tf.reduce_sum(-0.5 / tf.exp(2 * log_sigma) * tf.square(beta) -
                       log_sigma)
def regression_log_joint(beta, log_sigma, x, y):
  # This function returns log p(beta | log_sigma) + log p(y | x, beta).
  means = tf.matmul(tf.expand_dims(beta, 0), x, transpose_b=True)
  means = tf.squeeze(means)
  log_likelihood = tf.reduce_sum(-0.5 * tf.square(y - means))
  return log_prior(beta, log_sigma) + log_likelihood
def log_joint_partial(beta):
  return regression_log_joint(beta, log_sigma, x, y)
# Our estimate of log(sigma)
log_sigma = tf.Variable(0., name='log_sigma')
# The state of the Markov chain
beta = tf.Variable(tf.random_normal([x.shape[1]]), name='beta')
new_beta, _, _, _ = hmc.kernel(0.1, 5, beta, log_joint_partial,
                               event_dims=[0])
beta_update = tf.assign(beta, new_beta)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
with tf.control_dependencies([beta_update]):
  log_sigma_update = optimizer.minimize(-log_prior(beta, log_sigma),
                                        var_list=[log_sigma])

sess = tf.Session()
sess.run(tf.global_variables_initializer())
log_sigma_history = np.zeros(1000)
for i in xrange(1000):
  log_sigma_val, _ = sess.run([log_sigma, log_sigma_update])
  log_sigma_history[i] = log_sigma_val
# Should converge to something close to true_sigma
plt.plot(np.exp(log_sigma_history))
```


Last updated November 2, 2017.