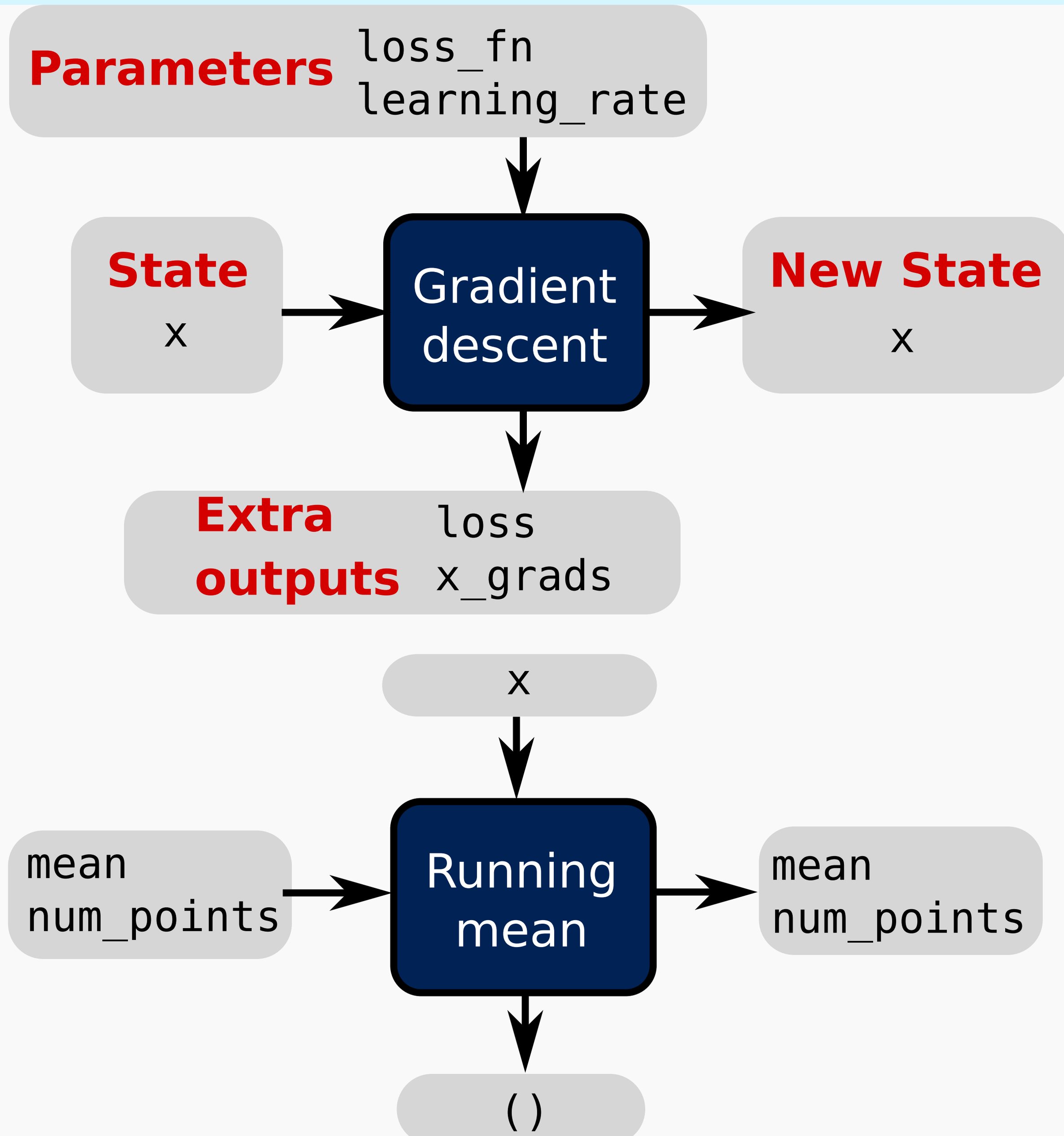
FunMC: A functional API for building Markov Chains

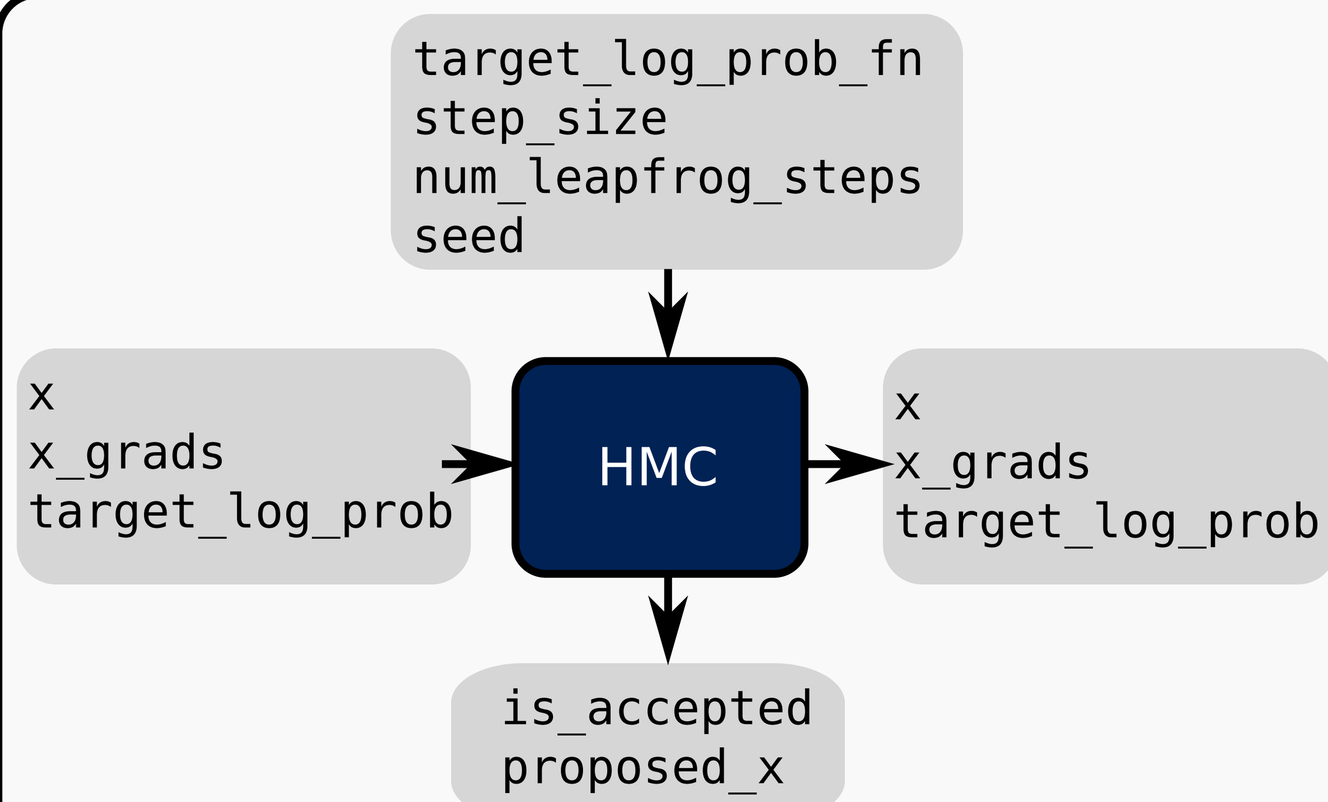
Pavel Sountsov, Alexey Radul, Srinivas Vasudevan
Google Research

FunMC is a TensorFlow/JAX library of utilities for accelerating methodological research into sequential, constant-memory algorithms like Markov Chain Monte Carlo (MCMC) and optimization. It does so by providing:

- MCMC building blocks
- Optimization operators
- Streaming statistics operators
- All with an API with minimal abstractions, favoring composition over configuration and utilization of the base language (Python) over a new DSL.

Core Abstraction: Transition Kernel





"Simplest" transition kernel and the `fun_mc.trace` transition kernel

Transition kernels are simply Python callables. fun_mc.trace is a workhorse utility used to advance and trace other transition kernels.

```
def transition_kernel(x):
  return x + 1, x

x_fin, x_trace = fun_mc.trace(state=0, fn=transition_kernel, num_steps=5)
x_fin  # => 5
x_trace # => [0, 1, 2, 3, 4]
```

Bayesian Logistic Regression with HMC

Target unnormalized log-densities are specified as callables that can also return extra outputs.

```
def target(w):
  logits = input_features @ w
  log_prob = normal_log_prob(w).sum(-1)
  log_prob += bernoulli_log_prob(outputs, logits[..., 0]).sum(-1)
  return log_prob, logits

def transition_kernel(hmc_state, key):
  hmc_key, key = jax.random.split(key)
  hmc_state, hmc_extra = fun_mc.hamiltonian_monte_carlo(hmc_state,
    target, step_size, num_integrator_steps, seed=hmc_key)
  w, logits = hmc_state.state, hmc_state.state_extra
  return (hmc_state, key), (w, logits, hmc_extra.is_accepted)

(fin_hmc_state, _), (w_chain, logits_chain, is_accepted_chain) = fun_mc.trace(
  (fun_mc.hamiltonian_monte_carlo_init(w_init, target), jax.random.PRNGKey(0)),
  transition_kernel, num_steps)
```

Reparameterization for constraints and preconditioning

Log-densities can be reparameterized using diffeomorphisms (implemented, e.g. in TensorFlow Probability).

```
reparam_potential_fn, reparam_w_init = fun_mc.reparameterize_potential_fn(
  target, diffeomorphism, w_init)

def transition_kernel(hmc_state, key):
  hmc_key, key = jax.random.split(key)
  hmc_state, hmc_extra = fun_mc.hamiltonian_monte_carlo(hmc_state,
    reparam_potential_fn, step_size, num_integrator_steps, seed=hmc_key)
  w, logits = hmc_state.state_extra[:2]
  return (hmc_state, key), (w, logits, hmc_extra.is_accepted)

_, (w_chain, logits_chain, is_accepted_chain) = fun_mc.trace(
  (fun_mc.hamiltonian_monte_carlo_init(reparam_w_init, reparam_potential_fn),
  jax.random.PRNGKey(0)), transition_kernel, num_steps)
```

HMC step size adaptation

Optimization can be combined with MCMC to produce adaptive MCMC algorithms.

```
def transition_kernel(hmc_state, log_step_size_state, key):
  hmc_key, key = jax.random.split(key)
  step_size = np.exp(log_step_size_state.state)
  hmc_state, hmc_extra = fun_mc.hamiltonian_monte_carlo(hmc_state,
    target, step_size, num_integrator_steps, seed=hmc_key)
  p_accept = np.exp(np.minimum(0., hmc_extra.log_accept_ratio))
  loss_fn = fun_mc.make_surrogate_loss_fn(lambda _: (0.8 - p_accept, ()))
  log_step_size_state, _ = fun_mc.adam_step(log_step_size_state, loss_fn,
    learning_rate=1e-2)
  w, logits = hmc_state.state, hmc_state.state_extra
  return (hmc_state, log_step_size_state, key), (w, logits, hmc_extra.is_accepted)
```

Streaming Statistics and Diagnostics

Streaming statistics enable analysis of an MCMC chain without materializing it.

```
def transition_kernel(hmc_state, cov_state, rhat_state, key):
  hmc_key, key = jax.random.split(key)
  hmc_state, hmc_extra = fun_mc.hamiltonian_monte_carlo(hmc_state,
    target, step_size, num_integrator_steps, seed=hmc_key)
  w, logits = hmc_state.state, hmc_state.state_extra
  cov_state, _ = fun_mc.running_covariance_step(cov_state, (w, logits), axis=0)
  rhat_state, _ = fun_mc.potential_scale_reduction_step(rhat_state, w)
  return (hmc_state, cov_state, rhat_state, key), ()

(_, fin_cov_state, fin_mean_accept_state, fin_rhat_state, _), _ = fun_mc.trace(
  (fun_mc.hamiltonian_monte_carlo_init(w_init, target),
   fun_mc.running_covariance_init((w_init.shape[-1:], y.shape), (np.float32,) * 2),
   fun_mc.potential_scale_reduction_init(w_init.shape, np.float32),
  jax.random.PRNGKey(0)), transition_kernel, num_steps)

w_cov, logits_cov = fin_cov_state.covariance
r_hat = fun_mc.potential_scale_reduction_extract(fin_rhat_state)
```

Markov Chain thinning

fun_mc.trace is itself a transition kernel, enabling chunked computation that can be used for thinning.

```
_, (w_chain, logits_chain, is_accepted_chain) = fun_mc.trace(
  (fun_mc.hamiltonian_monte_carlo_init(w_init, target), jax.random.PRNGKey(0)),
  lambda *state: fun_mc.trace(state, transition_kernel, num_substeps, trace_mask=False),
  num_steps // num_substeps)
```