BlackJax Tutorial

Bayesian Inference in JAX

January 29, 2026

What is BlackJax?

BlackJax is a library for sampling and inference in JAX

- High-performance Bayesian inference
- Built on JAX (GPU/TPU support)
- Multiple sampling algorithms (NUTS, HMC, MALA, etc.)
- Functional and composable design
- Automatic differentiation

Installation:

```
pip install blackjax
```

Why BlackJax?

Advantages

- **Fast**: JIT compilation
- **Flexible**: Customizable
- **Scalable**: GPU/TPU
- **Modular**: Mix & match
- **Interoperable**: Works with NumPyro

Use Cases

- Parameter estimation
- Probabilistic programming
- Scientific inference
- ML with uncertainty

Quick Start: Basic Example

```
import jax
import jax.numpy as jnp
import blackjax

def logprob_fn(x):
    return -0.5 * jnp.sum(x**2)

initial_position = jnp.array([1.0])
rng_key = jax.random.PRNGKey(0)
```

Step 1: Choose an Algorithm

```
nuts = blackjax.nuts(logprob_fn, step_size=0.1)

# Other algorithms available
```

Popular Algorithms:

- `nuts`: Adaptive HMC (best for complex posteriors)
- `hmc`: Hamiltonian Monte Carlo
- `mala`: Metropolis-Adjusted Langevin
- `rwm`: Random Walk Metropolis

Step 2: Initialize the Sampler

```
rng_key, init_key = jax.random.split(rng_key)
initial_state = nuts.init(initial_position)

print(f"Position: {initial_state.position}")
```

State contains:

- Current position
- Momentum (for HMC/NUTS)
- Acceptance rate
- Algorithm-specific info

Step 3: Run the Sampler

```
def inference_loop(rng_key, initial_state, num_samples):
    keys = jax.random.split(rng_key, num_samples)

    def one_step(state, key):
        new_state, info = nuts.step(key, state)
        return new_state, (new_state.position, info)

    final_state, (positions, infos) = jax.lax.scan(
        one_step, initial_state, keys
    )
    return positions, infos

num_samples = 1000
positions, infos = inference_loop(
    rng_key, initial_state, num_samples
)
```

Step 4: Analyze Results

```
import matplotlib.pyplot as plt

plt.plot(positions)
plt.xlabel('Iteration')
plt.ylabel('Parameter value')
plt.title('MCMC Trace')
plt.show()

mean_estimate = jnp.mean(positions[500:])
std_estimate = jnp.std(positions[500:])

print(f"Mean: {mean_estimate:.3f}")
print(f"Std: {std_estimate:.3f}")
```

Window Adaptation

Automatically tune step size and mass matrix:

```
from blackjax.adaptation import window_adaptation

adapt = window_adaptation(blackjax.nuts, logprob_fn)

(final_state, parameters), _ = adapt.run(
    rng_key, initial_position, num_steps=1000
)

adapted_nuts = blackjax.nuts(
    logprob_fn,
    step_size=parameters['step_size'],
    inverse_mass_matrix=parameters['inverse_mass_matrix']
)
```

Multi-Dimensional Problems

```
def logprob_2d(x):
    mu = jnp.array([0.0, 0.0])
    sigma = jnp.array([[1.0, 0.5],
                       [0.5, 1.0]])
    diff = x - mu
    return -0.5 * diff @ jnp.linalg.inv(sigma) @ diff

initial_position = jnp.array([1.0, 1.0])

nuts = blackjax.nuts(logprob_2d, step_size=0.1)
initial_state = nuts.init(initial_position)
```

Real Example: Linear Regression (1/2)

```
true_slope = 2.0
true_intercept = 1.0
x_data = jnp.linspace(0, 10, 50)
noise = jax.random.normal(jax.random.PRNGKey(0), (50,))
y_data = true_slope * x_data + true_intercept + noise * 0.5
```

Real Example: Linear Regression (2/2)

```
def logprob_regression(params):
    slope, intercept, log_sigma = params
    sigma = jnp.exp(log_sigma)

    y_pred = slope * x_data + intercept
    log_likelihood = jnp.sum(
        -0.5 * ((y_data - y_pred) / sigma)**2
        - jnp.log(sigma)
    )

    log_prior = (-0.5 * slope**2
                 - 0.5 * intercept**2
                 - 0.5 * log_sigma**2)

    return log_likelihood + log_prior
```

Linear Regression: Inference

```
initial_params = jnp.array([0.0, 0.0, 0.0])

adapt = window_adaptation(
    blackjax.nuts, logprob_regression
)
(final_state, parameters), _ = adapt.run(
    rng_key, initial_params, num_steps=1000
)

nuts = blackjax.nuts(
    logprob_regression,
    step_size=parameters['step_size'],
    inverse_mass_matrix=parameters['inverse_mass_matrix']
)

samples, _ = inference_loop(
    sample_key, final_state, 2000
)
```

Vectorized Sampling (Multiple Chains)

```
num_chains = 4

rng_keys = jax.random.split(rng_key, num_chains)
initial_positions = jax.random.normal(
    rng_keys[0], (num_chains, 3)
)

vmapped_step = jax.vmap(
    lambda key, state: nuts.step(key, state),
    in_axes=(0, 0)
)

initial_states = jax.vmap(nuts.init)(initial_positions)
```

Performance Tips

1. **JIT Compilation**
   - Use `jax.jit` for speed
2. **GPU Acceleration**
   - Automatically uses GPU if available
3. **Batching**
   - Use `vmap` for multiple chains
4. **Step Size Tuning**
   - Use `window_adaptation`
   - Target $\sim$65% for NUTS

Common Algorithms

Algorithm	When to Use	Key Parameters
NUTS	Default choice	`step_size`
HMC	Known structure	`step_size`, `num_steps`
MALA	High dimensions	`step_size`
RWM	Simple problems	`step_size`

Diagnostics

Key Metrics:

- **R-hat**: Gelman-Rubin convergence diagnostic
- **ESS**: Effective sample size
- **Acceptance rate**: Should be $\sim$65% for NUTS
- **Trace plots**: Visual inspection
- **Autocorrelation**: Check mixing

```
acceptance_rate = jnp.mean(
    infos.acceptance_probability
)
print(f"Accept rate: {acceptance_rate:.2f}")
```

Best Practices

1. Start simple - test on small problems first
2. Use adaptation - let BlackJax tune parameters
3. Multiple chains - run 4+ chains to check convergence
4. Burn-in - discard initial samples (e.g., first 50%)
5. Check diagnostics - R-hat, ESS, trace plots
6. Profile code - use JAX profiler for large problems
7. Batch data - for big datasets, use mini-batching

Resources

Documentation:

- Official Docs: `https://blackjax-devs.github.io/blackjax/`
- GitHub: `https://github.com/blackjax-devs/blackjax`

Related Libraries:

- NumPyro: Probabilistic programming
- JAX: Autodiff and JIT
- Optax: Optimization

Key Papers:

- Hoffman & Gelman (2014): The No-U-Turn Sampler
- Neal (2011): MCMC using Hamiltonian dynamics

Quick Reference

```
import blackjax
import jax

def logprob(x):
    return -0.5 * jnp.sum(x**2)

adapt = blackjax.window_adaptation(
    blackjax.nuts, logprob
)
(state, params), _ = adapt.run(
    rng_key, init_pos, num_steps=1000
)

sampler = blackjax.nuts(logprob, **params)

keys = jax.random.split(rng_key, 1000)
_, samples = jax.lax.scan(step, state, keys)

posterior_mean = jnp.mean(samples.position, axis=0)
```

Summary

BlackJax makes Bayesian inference fast, flexible, and fun!

Thank you!

Questions?