Lesson 2: MCMC Sampling and Inference

Learning Objectives

By the end of this lesson, you will be able to:

- Understand MCMC fundamentals in PyMC
- Learn about different samplers (NUTS, Metropolis)
- Master trace analysis and convergence diagnostics
- Handle sampling issues and optimize performance

Prerequisites

- Completed Lesson 1
- Understanding of Markov chains (helpful but not required)

1. MCMC Fundamentals

Markov Chain Monte Carlo (MCMC) is the engine behind PyMC's posterior inference. Instead of computing the posterior analytically, MCMC generates samples that approximate the posterior distribution.

Key Concepts:

- **Markov Chain**: Sequence of samples where each sample depends only on the previous one
- **Stationary Distribution**: The target posterior distribution
- **Burn-in**: Initial samples discarded to ensure convergence
- **Thinning**: Taking every nth sample to reduce autocorrelation

```
import pymc as pm
import numpy as np
import matplotlib.pyplot as plt
import arviz as az

# Set up for reproducibility
np.random.seed(42)
```

2. Sampling in PyMC

Basic Sampling

```
# Simple example: estimating mean of normal distribution
true_mu = 5.0
true_sigma = 2.0
n_obs = 100

# Generate data
data = np.random.normal(true_mu, true_sigma, n_obs)

with pm.Model() as simple_model:
    # Prior
    mu = pm.Normal('mu', mu=0, sigma=10)
    sigma = pm.HalfNormal('sigma', sigma=5)

    # Likelihood
    obs = pm.Normal('obs', mu=mu, sigma=sigma, observed=data)

    # Sample with different configurations
    trace = pm.sample(
        draws=2000,           # Number of samples after tuning
        tune=1000,            # Number of tuning samples
        chains=4,             # Number of parallel chains
        cores=2,              # Number of CPU cores to use
        return_inferencedata=True
    )
```

Sampler Configuration

```
# Different samplers and their use cases
with pm.Model() as model:
    mu = pm.Normal('mu', mu=0, sigma=10)
    sigma = pm.HalfNormal('sigma', sigma=5)
    obs = pm.Normal('obs', mu=mu, sigma=sigma, observed=data)

    # NUTS (default) - good for continuous variables
    trace_nuts = pm.sample(1000, step=pm.NUTS())

    # Metropolis - for discrete or difficult continuous variables
    trace_metropolis = pm.sample(1000, step=pm.Metropolis())

    # Slice sampler - good for some univariate distributions
    trace_slice = pm.sample(1000, step=pm.Slice())
```

3. Convergence Diagnostics

R-hat Statistic

R-hat measures chain mixing by comparing within-chain and between-chain variance:

```
# Check R-hat values
rhat = az.rhat(trace)
print("R-hat values:")
print(rhat)

# R-hat interpretation:
# < 1.01: Excellent convergence
# 1.01-1.05: Good convergence
# 1.05-1.1: Acceptable convergence
# > 1.1: Poor convergence - need more samples
```

Effective Sample Size (ESS)

```
# Check effective sample size
ess = az.ess(trace)
print("Effective Sample Size:")
print(ess)

# ESS interpretation:
# > 400: Generally sufficient
# 100-400: May be adequate depending on the analysis
# < 100: Likely insufficient
```

Trace Plots

```
# Visualize convergence
az.plot_trace(trace, var_names=['mu', 'sigma'])
plt.suptitle("Trace Plots - Check for Convergence")
plt.show()

# What to look for:
# - Stationary behavior (no trends)
# - Good mixing between chains
# - No stuck chains
# - Reasonable coverage of parameter space
```

4. Advanced Sampling Techniques

Handling Difficult Sampling Situations

```
# Example: Highly correlated parameters
with pm.Model() as correlated_model:
    # Create correlated priors
    x = pm.Normal('x', mu=0, sigma=1)
    y = pm.Normal('y', mu=x, sigma=0.1)  # y highly correlated with x
    # This might sample poorly with default settings
    trace_default = pm.sample(1000)

    # Better approach: increase target_accept
    trace_better = pm.sample(1000, target_accept=0.95)
```

Custom Step Methods

```
# Combining different step methods
with pm.Model() as mixed_model:
    # Continuous variable
    continuous_var = pm.Normal('continuous', mu=0, sigma=1)

    # Discrete variable
    discrete_var = pm.Binomial('discrete', n=10, p=0.5)

    # Mixed sampling
    step1 = pm.NUTS([continuous_var])
    step2 = pm.BinaryGibbsMetropolis([discrete_var])

    trace = pm.sample(1000, step=[step1, step2])
```

5. Practical Example: Robust Regression

Let's implement a robust regression model that's challenging to sample:

```
# Generate data with outliers
n = 100
x = np.linspace(0, 10, n)
y = 2 + 0.5 * x + np.random.normal(0, 0.5, n)

# Add some outliers
outlier_idx = np.random.choice(n, size=5, replace=False)
y[outlier_idx] += np.random.normal(0, 5, 5)

# Standard linear regression (for comparison)
with pm.Model() as normal_model:
    alpha = pm.Normal('alpha', mu=0, sigma=10)
    beta = pm.Normal('beta', mu=0, sigma=10)
    sigma = pm.HalfNormal('sigma', sigma=1)

    mu = alpha + beta * x
    likelihood = pm.Normal('y', mu=mu, sigma=sigma, observed=y)

    trace_normal = pm.sample(2000, tune=1000)

# Robust regression with Student-t likelihood
with pm.Model() as robust_model:
    alpha = pm.Normal('alpha', mu=0, sigma=10)
    beta = pm.Normal('beta', mu=0, sigma=10)
    sigma = pm.HalfNormal('sigma', sigma=1)
    nu = pm.Exponential('nu', lam=1/30)  # Degrees of freedom
    mu = alpha + beta * x
    likelihood = pm.StudentT('y', nu=nu, mu=mu, sigma=sigma, observed=y)

    # This model is more challenging to sample
    trace_robust = pm.sample(2000, tune=1000, target_accept=0.90)
```

6. Sampling Diagnostics and Troubleshooting

Common Sampling Issues

```
# Check for sampling problems
def diagnose_sampling(trace):
    """Comprehensive sampling diagnostics"""
    print("=== Sampling Diagnostics ===")

    # Check R-hat
    rhat = az.rhat(trace)
    print(f"Max R-hat: {rhat.max().values:.4f}")
    if rhat.max() > 1.1:
        print("⚠  WARNING: Poor convergence detected!")

    # Check ESS
    ess = az.ess(trace)
    min_ess = ess.min()
    print(f"Min ESS: {min_ess.values:.0f}")
    if min_ess < 100:
        print("⚠  WARNING: Low effective sample size!")

    # Check divergences
    divergences = trace.sample_stats.diverging.sum()
    print(f"Number of divergences: {divergences.values}")
    if divergences > 0:
        print("⚠  WARNING: Divergent transitions detected!")

    # Energy diagnostics
    energy = trace.sample_stats.energy
    az.plot_energy(trace)
    plt.title("Energy Plot - Check for Sampling Issues")
    plt.show()

# Apply diagnostics
diagnose_sampling(trace_robust)
```

Fixing Sampling Problems

```
# Solutions for common issues:

# 1. Divergent transitions → increase target_accept
with pm.Model() as fixed_model:
    # ... model specification ...
    trace_fixed = pm.sample(2000, target_accept=0.95)
```

```
# 2. Poor mixing → reparameterization
with pm.Model() as reparam_model:
    # Instead of: sigma = pm.HalfNormal('sigma', sigma=1)
    # Use:
    sigma_raw = pm.Normal('sigma_raw', mu=0, sigma=1)
    sigma = pm.Deterministic('sigma', pm.math.exp(sigma_raw))

# 3. Multimodal posteriors → longer chains
with pm.Model() as long_chain_model:
    # ... model specification ...
    trace_long = pm.sample(10000, tune=5000)
```

7. Exercises

Exercise 1: Sampling Comparison

Compare different samplers on a simple model:

```
# Create a model with known posterior
with pm.Model() as comparison_model:
    theta = pm.Beta('theta', alpha=2, beta=3)
    obs = pm.Binomial('obs', n=20, p=theta, observed=15)

    # Sample with different methods
    trace_nuts = pm.sample(1000, step=pm.NUTS())
    trace_metropolis = pm.sample(1000, step=pm.Metropolis())

    # Compare efficiency and accuracy
    # [Your analysis here]
```

Exercise 2: Convergence Analysis

Create a model that initially has convergence issues:

```
# Highly correlated parameters
with pm.Model() as difficult_model:
    x = pm.Normal('x', mu=0, sigma=1)
    y = pm.Normal('y', mu=x, sigma=0.01)  # Very tight correlation

    # Try different sampling strategies
    # [Your implementation here]
```

Exercise 3: Custom Diagnostics

Implement your own convergence diagnostic:

```
def autocorrelation_diagnostic(trace, var_name, max_lag=50):
    """
    Compute autocorrelation for a trace
    """
    # Extract samples
    samples = trace.posterior[var_name].values.flatten()

    # Compute autocorrelation
    # [Your implementation here]

    return autocorrelations

# Apply to your traces
```

8. Key Takeaways

1. **MCMC is Approximation**: Samples approximate the posterior, not exact values
2. **Convergence is Critical**: Always check R-hat and ESS before interpretation
3. **Multiple Chains**: Use multiple chains to detect convergence issues
4. **Tune Parameters**: Adjust target_accept and other parameters for difficult models
5. **Diagnostic Plots**: Visual inspection is as important as numerical diagnostics

9. Common Pitfalls

1. **Ignoring Diagnostics**: Never skip convergence checks
2. **Too Few Samples**: Ensure adequate ESS for your analysis
3. **Single Chain**: Always use multiple chains
4. **Default Settings**: Don't always trust default sampler settings
5. **Divergent Transitions**: Address divergences before proceeding

10. Next Steps

In the next lesson, we'll explore prior and posterior predictive checks, which are crucial for model validation and the complete Bayesian workflow. You'll learn how to check if your model makes sense before and after seeing the data.

11. Further Reading

- MCMC in PyMC
- Convergence Diagnostics
- Energy Plots

Time to complete: 2-3 hours
Difficulty: Intermediate
Prerequisites: Lesson 1, Basic MCMC concepts