Particle Filter and Differentiable Particle Filter

November 15, 2025

Abstract

This guide provides a practical, hands-on approach to understanding particle filters (PF) and differentiable particle filters (DPF). We follow the philosophy of "getting your hands dirty first" – building intuition through implementation before diving into theoretical proofs. The guide covers the mathematical foundations, algorithmic details, and complete Python implementations of both standard and differentiable particle filters.

Contents

1 Introduction

This guide provides a practical approach to understanding particle filters and their differentiable variants. The key insight is to start with implementation and visualization before exploring the underlying theory.

1.1 Learning Path

1. Start with basic particle filter implementation
2. Understand the mathematical framework
3. Transition to differentiable versions
4. Apply to learning problems with gradient descent

2 Prerequisites

2.1 Required Knowledge

- Basic probability theory (Bayes' rule, conditional distributions)
- Python programming
- Basic linear algebra
- Familiarity with NumPy and PyTorch

2.2 Software Requirements

The following Python packages are required:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
```

3 Mathematical Foundation

3.1 State Space Model (SSM)

The generative model assumes a Hidden Markov Model (HMM) structure with the following components:

State Space Model

State Transition:

$$x_t = f(x_{t-1}) + q_t, \quad q_t \sim \mathcal{N}(0, Q) \tag{1}$$

Observation:

$$y_t = g(x_t) + r_t, \quad r_t \sim \mathcal{N}(0, R) \tag{2}$$

3.1.1 Toy Example Specification

In our illustrative example:

- **State transition:** $x_t = 0.9x_{t-1} + \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0, 0.5^2)$
- **Observation:** $y_t = x_t + \eta_t$, where $\eta_t \sim \mathcal{N}(0, 0.3^2)$

Goal: Estimate the posterior distribution $p(x_t \mid y_{1:t})$

3.2 Bayesian Recursion

The optimal Bayesian filtering solution follows a two-step recursion:

Bayesian Filter

Prediction Step:

$$p(x_t \mid y_{1:t-1}) = \int p(x_t \mid x_{t-1}) p(x_{t-1} \mid y_{1:t-1})\, dx_{t-1} \tag{3}$$

Update Step:

$$p(x_t \mid y_{1:t}) \propto p(y_t \mid x_t) p(x_t \mid y_{1:t-1}) \tag{4}$$

Remark 3.1. This integral is intractable for nonlinear functions f and g. The particle filter provides a Monte Carlo approximation to this recursion.

4 Standard Particle Filter

4.1 Core Idea

Approximate the posterior distribution with weighted samples (particles):

$$p(x_t \mid y_{1:t}) \approx \sum_{i=1}^{N} w_t^{(i)} \delta(x_t - x_t^{(i)}) \tag{5}$$

where:

- $x_t^{(i)}$: particle i at time t
- $w_t^{(i)}$: normalized weight of particle i
- N: total number of particles
- $\delta(\cdot)$: Dirac delta function

4.2 Algorithm

Particle Filter Algorithm

Initialization: For $i = 1, \ldots, N$

- Sample $x_0^{(i)} \sim p(x_0)$
- Set $w_0^{(i)} = 1/N$

For each time step $t = 1, 2, \ldots, T$:

1. **Prediction:** For $i = 1, \ldots, N$

$$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)}) = \mathcal{N}(f(x_{t-1}^{(i)}), Q) \tag{6}$$

2. **Weight Update:** For $i = 1, \ldots, N$

$$\tilde{w}_t^{(i)} = w_{t-1}^{(i)} \cdot p(y_t \mid x_t^{(i)}) \tag{7}$$

where the likelihood is:

$$p(y_t \mid x_t^{(i)}) = \frac{1}{\sqrt{2\pi R}} \exp\left(-\frac{(y_t - g(x_t^{(i)}))^2}{2R}\right) \tag{8}$$

3. **Normalization:** For $i = 1, \ldots, N$

$$w_t^{(i)} = \frac{\tilde{w}_t^{(i)}}{\sum_{j=1}^{N} \tilde{w}_t^{(j)}} \tag{9}$$

4. **State Estimation:**

$$\hat{x}_t = \sum_{i=1}^{N} w_t^{(i)} x_t^{(i)} \tag{10}$$

5. **Resampling:** For $i = 1, \ldots, N$
   - Draw $a_i \sim \text{Categorical}(w_t^{(1)}, \ldots, w_t^{(N)})$
   - Set $x_t^{(i)} \leftarrow x_t^{(a_i)}$
   - Reset $w_t^{(i)} \leftarrow 1/N$

4.3 Why Resampling?

Remark 4.1. Without resampling, most particles will have negligible weights after several iterations – a phenomenon known as **particle degeneracy**. Resampling concentrates computational resources (particles) in high-probability regions of the state space.

5 Differentiable Particle Filter

5.1 Motivation

Problem with Standard PF:

- The resampling step uses discrete sampling: $a_i \sim \text{Categorical}(w_t)$
- This operation is **non-differentiable**
- Cannot compute gradients $\frac{\partial \mathcal{L}}{\partial \theta}$
- Cannot use gradient descent to learn parameters $\theta = \{f_\theta, g_\theta, Q_\theta, R_\theta\}$

Solution: Replace discrete resampling with a **soft, differentiable approximation**.

5.2 Key Modification: Soft Resampling via Gumbel-Softmax

Replace categorical sampling with the **Gumbel-Softmax** trick:

$$\tilde{w}_t^{(i)} = \frac{\exp\left((\log w_t^{(i)} + g^{(i)})/\tau\right)}{\sum_{j=1}^{N} \exp\left((\log w_t^{(j)} + g^{(j)})/\tau\right)} \tag{11}$$

where:

- $g^{(i)} \sim \text{Gumbel}(0, 1)$ are i.i.d. Gumbel random variables
- $\tau > 0$ is the temperature parameter
- As $\tau \to 0$: recovers hard (categorical) resampling
- For $\tau > 0$: the operation becomes differentiable

Gumbel Sampling in Code:

```
        g = -torch.log(-torch.log(torch.rand_like(weights) +
            eps) + eps)
```

5.3 Gradient Flow and End-to-End Learning

Since all operations are now differentiable, we can compute:

$$\frac{\partial \mathcal{L}}{\partial \theta} \tag{12}$$

via backpropagation, where:

- θ: learnable parameters in f_θ, g_θ, Q_θ, R_θ
- $\mathcal{L} = \sum_{t=1}^{T} \|y_t - \hat{y}_t\|_2^2$ or other task-specific loss

This enables **end-to-end learning** of latent dynamics and observation models.

6 Hands-On Implementation

6.1 Part 1: Standard Particle Filter

```python
import numpy as np
import matplotlib.pyplot as plt

# Setup
T = 50   # Time steps
N = 1000 # Number of particles

# Generate true hidden states and observations
true_x = np.zeros(T)
y = np.zeros(T)
true_x[0] = 0

for t in range(1, T):
true_x[t] = 0.9 * true_x[t-1] + np.random.randn() * 0.5

y = true_x + np.random.randn(T) * 0.3

# Initialize particles
particles = np.random.randn(N)
weights = np.ones(N) / N
estimates = []

# Main filter loop
for t in range(T):
# Prediction step
particles = 0.9 * particles + np.random.randn(N) * 0.5

# Weight update
likelihood = np.exp(-0.5 * ((y[t] - particles) / 0.3)**2)
weights *= likelihood
weights += 1e-300  # Avoid zeros
weights /= np.sum(weights)

# Estimate
est = np.sum(particles * weights)
estimates.append(est)

# Resample
idx = np.random.choice(N, N, p=weights)
particles = particles[idx]
weights.fill(1.0 / N)

# Visualization
plt.figure(figsize=(12, 6))
plt.plot(true_x, label="True State", linewidth=2)
```

```
        plt.plot(y, label="Observations", alpha=0.5)
        plt.plot(estimates, label="PF Estimate", linewidth=2)
        plt.legend()
        plt.xlabel("Time")
        plt.ylabel("Value")
        plt.title("Particle Filter Performance")
        plt.grid(True)
        plt.show()
```

Expected Output:

- The particle filter estimate closely tracks the true hidden state
- The estimate is smoother than raw observations

6.2 Part 2: Differentiable Particle Filter

```
import torch
import torch.nn as nn

def soft_resample(weights, eps=1e-8, temperature=0.1):
"""Differentiable resampling using Gumbel-Softmax"""
# Sample Gumbel noise
g = -torch.log(-torch.log(torch.rand_like(weights) +
    eps) + eps)

# Add noise to log weights
logits = torch.log(weights + eps) + g

# Apply softmax
return torch.nn.functional.softmax(logits /
    temperature, dim=0)

# Setup
N = 100  # Fewer particles for faster training
T = 20

# Generate data
x = torch.zeros(T)
x[0] = 0.
for t in range(1, T):
x[t] = 0.9 * x[t-1] + torch.randn(1) * 0.5

y = x + torch.randn(T) * 0.3

# Initialize particles
particles = torch.randn(N, requires_grad=False)
weights = torch.ones(N) / N
estimates = []

# Main filter loop
```

```python
for t in range(T):
# Prediction
particles = 0.9 * particles + torch.randn(N) * 0.5

# Weight update
likelihood = torch.exp(-0.5 * ((y[t] - particles) /
    0.3)**2)
weights = weights * likelihood
weights = weights / weights.sum()

# Soft resampling (differentiable!)
weights = soft_resample(weights)

# Estimate
est = torch.sum(particles * weights)
estimates.append(est)

# Define loss and compute gradients
estimates_tensor = torch.stack(estimates)
loss = torch.mean((y - estimates_tensor)**2)
# loss.backward()  # Gradients can flow through!

print(f"MSE Loss: {loss.item():.4f}")
```

6.3 Part 3: Learning System Parameters

```python
import torch
import torch.optim as optim

class LearnableSSM(nn.Module):
"""State Space Model with learnable transition
    coefficient"""
def __init__(self):
super().__init__()
# Learnable parameter (initialized incorrectly)
self.transition_coef = nn.Parameter(torch.tensor(0.5))
self.process_noise = 0.5
self.obs_noise = 0.3

def forward(self, y, N=100, T=None):
if T is None:
T = len(y)

# Initialize
particles = torch.randn(N)
weights = torch.ones(N) / N
estimates = []

for t in range(T):
# Prediction with learnable coefficient
```

```
particles = self.transition_coef * particles + \
torch.randn(N) * self.process_noise

# Update
likelihood = torch.exp(-0.5 * ((y[t] - particles) /
self.obs_noise)**2)
weights = weights * likelihood
weights = weights / weights.sum()

# Soft resample
g = -torch.log(-torch.log(torch.rand_like(weights) + 1e-8) + 1e-8)
logits = torch.log(weights + 1e-8) + g
weights = torch.nn.functional.softmax(logits / 0.1, dim=0)

# Estimate
est = torch.sum(particles * weights)
estimates.append(est)

return torch.stack(estimates)

# Generate training data (true coefficient = 0.9)
T_train = 50
x_true = torch.zeros(T_train)
for t in range(1, T_train):
x_true[t] = 0.9 * x_true[t-1] + torch.randn(1) * 0.5
y_train = x_true + torch.randn(T_train) * 0.3

# Training
model = LearnableSSM()
optimizer = optim.Adam(model.parameters(), lr=0.01)

print(f"Initial: {model.transition_coef.item():.4f}")

for epoch in range(100):
optimizer.zero_grad()
x_pred = model(y_train)
loss = torch.mean((y_train - x_pred)**2)
loss.backward()
optimizer.step()

if (epoch + 1) % 20 == 0:
print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}, "
f"Coef: {model.transition_coef.item():.4f}")

print(f"\nFinal: {model.transition_coef.item():.4f}")
print(f"True: 0.9")
```

Expected Result: The learned coefficient converges toward the true value of 0.9.

7 Advanced Topics

7.1 Particle Degeneracy

Problem: After several iterations, most particle weights become negligible.
Solutions:

- Standard resampling (for classical PF)
- Soft resampling (for DPF)
- Adaptive number of particles
- Regularization techniques
- Monitoring effective sample size: $\text{ESS} = 1/\sum_{i=1}^{N}(w_t^{(i)})^2$

7.2 Nonlinear Dynamics

Extend to nonlinear systems:

```
        # Example: Sine transition
        def nonlinear_transition(x, noise_std=0.5):
        return torch.sin(x) + torch.randn_like(x) * noise_std

        # Use in forward pass
        particles = nonlinear_transition(particles)
```

7.3 Comparison with Other Methods

Table 1: Comparison of Filtering Methods

Method	Pros	Cons
Extended Kalman Filter	Fast, analytical	Fails for strong nonlinearity
Particle Filter	Handles any nonlinearity	Not differentiable
Differentiable PF	Learnable, handles nonlinearity	Computationally expensive

8 Summary

Table 2: Standard vs Differentiable Particle Filter

Aspect	Standard PF	Differentiable PF
Transition	$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)})$	Same
Likelihood	$p(y_t \mid x_t^{(i)})$	Same
Weight Update	Hard normalization	Soft normalization
Resampling	Categorical sampling	Gumbel-Softmax
Gradient	Blocked	Flows via backprop
Purpose	State estimation	Learnable inference

A Gumbel Distribution

The Gumbel distribution enables differentiable sampling from categorical distributions.

A.1 Gumbel-Max Trick

If $g_i \sim \text{Gumbel}(0, 1)$ independently, then:

$$\arg\max_i (\log \pi_i + g_i) \sim \text{Categorical}(\pi) \tag{13}$$

A.2 Gumbel-Softmax Relaxation

Replace $\arg\max$ with softmax for differentiability:

$$\tilde{\pi}_i = \frac{\exp((\log \pi_i + g_i)/\tau)}{\sum_j \exp((\log \pi_j + g_j)/\tau)} \tag{14}$$

This provides a continuous, differentiable approximation to categorical sampling.