HDMR-PRO-SEN

High Dimensional Model Representation and Enhanced Multivariate Products Representation

A powerful Python library for tensor decomposition using HDMR and EMPR methods with multi-backend support (NumPy, PyTorch, TensorFlow).

🎯 What is HDMR-PRO-SEN?

HDMRLib implements two state-of-the-art tensor decomposition methods:

- **HDMR (High Dimensional Model Representation)**: Decomposes multivariate functions into hierarchical components with weighted support vectors
- **EMPR (Enhanced Multivariate Products Representation)**: An optimized variant using unweighted support vectors for better computational efficiency

Both methods represent complex multivariate functions as sums of lower-dimensional components, making them ideal for:

- **Sensitivity Analysis** - Identify which variables matter most
- **Uncertainty Quantification** - Understand how input uncertainties propagate
- **Function Approximation** - Approximate complex functions with simpler components
- **Dimensionality Reduction** - Reduce computational complexity while preserving accuracy

---

🚀 Quick Start

Installation

1. **Clone the repository:**

```
git clone https://github.com/your-username/HDMR-PRO-SEN.git
cd HDMR-PRO-SEN
```

2. **Create virtual environment:**

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install dependencies:**

```
pip install -r requirements.txt
```

GPU Acceleration (Optional)

For CUDA/GPU support:

```
# PyTorch with CUDA
pip install torch --index-url https://download.pytorch.org/whl/cu118  # CUDA 11.8

# TensorFlow (GPU support included)
pip install tensorflow
```

Requirements:

- NVIDIA GPU with CUDA capability
- CUDA Toolkit (11.8 or 12.1)
- cuDNN library

Basic Usage

```python
import numpy as np
from hdmr import HDMR
from empr import EMPR
from backends import set_backend

# Create test data
tensor = np.random.rand(5, 5, 5)

# Set backend (numpy, torch, tensorflow)
set_backend('numpy')

# EMPR Decomposition
empr_model = EMPR(tensor)
empr_result = empr_model.decompose(order=2)
empr_components = empr_model.components(max_order=2)

# HDMR Decomposition
hdmr_model = HDMR(tensor)
hdmr_result = hdmr_model.decompose(order=2)
hdmr_components = hdmr_model.components(max_order=2)

print(f"EMPR MSE: {np.mean((tensor - empr_result) ** 2):.6e}")
print(f"Available components: {list(empr_components.keys())}")
```

📚 Detailed Usage Guide

1. **Backend Selection**

HDMR-PRO-SEN supports multiple computational backends:

```python
from backends import set_backend, get_backend

# Available backends
set_backend('numpy')      # NumPy (default, always available)
set_backend('torch')      # PyTorch (requires torch>=2.2)
set_backend('tensorflow') # TensorFlow (requires tensorflow>=2.14)

# Check current backend
print(f"Current backend: {get_backend()}")
```

2. **EMPR (Enhanced Multivariate Products Representation)**

EMPR provides efficient decomposition with configurable support vectors:

```python
import numpy as np
from empr import EMPR

# Create test tensor
x, y, z = np.meshgrid(np.linspace(0, 1, 6), np.linspace(0, 1, 6), np.linspace(0,
1, 6), indexing='ij')
tensor = 1 + 2*x + 3*y + 4*z + x*y + x*z + y*z

# Initialize EMPR with different support vectors
empr_das = EMPR(tensor, supports='das')     # Data-Adaptive Supports (recommended)
empr_ones = EMPR(tensor, supports='ones')   # Uniform supports

# Decompose at different orders
result_order1 = empr_das.decompose(order=1)  # Main effects only
result_order2 = empr_das.decompose(order=2)  # Main effects + 2-way interactions
result_order3 = empr_das.decompose(order=3)  # Full decomposition

# Extract components
components = empr_das.components(max_order=3)
print("Available components:", list(components.keys()))
# Output: ['g1', 'g2', 'g3', 'g12', 'g13', 'g23', 'g123']

# Component interpretation:
# g1, g2, g3: Main effects (univariate)
# g12, g13, g23: Two-way interactions (bivariate)
# g123: Three-way interaction (trivariate)
```

3. **HDMR (High Dimensional Model Representation)**

HDMR uses weighted support vectors for enhanced numerical stability:

```python
from hdmr import HDMR

# Initialize HDMR with different weight types
```

```
hdmr_avg = HDMR(tensor, weight='avg')         # Average weights (default)
hdmr_gauss = HDMR(tensor, weight='gaussian') # Gaussian weights
hdmr_cheb = HDMR(tensor, weight='chebyshev') # Chebyshev weights

# Custom weights
custom_weights = [np.ones((6, 1)), np.ones((6, 1)), np.ones((6, 1))]
hdmr_custom = HDMR(tensor, weight='custom', custom_weights=custom_weights)

# Decompose and analyze
result = hdmr_avg.decompose(order=2)
components = hdmr_avg.components(max_order=2)

# Calculate reconstruction quality
mse = np.mean((tensor - result) ** 2)
print(f"HDMR Reconstruction MSE: {mse:.6e}")
```

4. **Custom Support Vectors**

For advanced users, you can provide custom support vectors:

```
# Define custom support vectors for each dimension
custom_supports = [
    np.linspace(0, 1, 6).reshape(-1, 1),  # Linear support for dimension 1
    np.exp(np.linspace(-1, 1, 6)).reshape(-1, 1),  # Exponential for dimension 2
    np.sin(np.linspace(0, np.pi, 6)).reshape(-1, 1)  # Sinusoidal for dimension 3
]

# Use custom supports
empr_custom = EMPR(tensor, supports='custom', custom_supports=custom_supports)
result = empr_custom.decompose(order=2)
```

5. **Multi-Backend Comparison**

Compare performance across different backends:

```
import time
from backends import set_backend

backends = ['numpy', 'torch', 'tensorflow']
tensor = np.random.rand(8, 8, 8)

for backend in backends:
    try:
        set_backend(backend)

        start_time = time.time()
        model = EMPR(tensor)
        result = model.decompose(order=2)
        elapsed = time.time() - start_time
```

```
        mse = np.mean((tensor - result) ** 2)
        print(f"{backend:>12}: {elapsed:.4f}s, MSE: {mse:.6e}")

    except Exception as e:
        print(f"{backend:>12}: Error - {e}")
```

🔬 Advanced Examples

Sensitivity Analysis

Analyze which variables contribute most to function variation:

```
# Create function with known variable importance
x1, x2, x3 = np.meshgrid(np.linspace(-1, 1, 8), np.linspace(-1, 1, 8),
np.linspace(-1, 1, 8), indexing='ij')
tensor = 5*x1**2 + 2*x2 + 0.1*x3 + x1*x2  # x1 most important, x3 least

# Decompose with EMPR
model = EMPR(tensor, supports='das')
components = model.components(max_order=2)

# Calculate component variances (sensitivity indicators)
sensitivities = {}
for comp_name, comp_tensor in components.items():
    sensitivities[comp_name] = np.var(comp_tensor)

# Sort by importance
sorted_sens = sorted(sensitivities.items(), key=lambda x: x[1], reverse=True)
print("Component sensitivities (most to least important):")
for comp, sens in sorted_sens:
    print(f"  {comp}: {sens:.4f}")
```

Function Approximation

Use HDMR/EMPR for efficient function approximation:

```
# Original expensive function
def expensive_function(x, y, z):
    return np.sin(np.pi*x) * np.cos(np.pi*y) * np.exp(z) + x*y*z

# Create training data
x, y, z = np.meshgrid(np.linspace(0, 1, 10), np.linspace(0, 1, 10), np.linspace(0,
1, 10), indexing='ij')
training_data = expensive_function(x, y, z)

# Train EMPR surrogate
surrogate = EMPR(training_data, supports='das')
```

```python
# Test approximation quality at different orders
for order in [1, 2, 3]:
    approx = surrogate.decompose(order=order)

    # Compare on training grid
    error = np.mean((training_data - approx) ** 2)
    print(f"Order {order} approximation MSE: {error:.6e}")
```

Performance Optimization

Optimize performance for large tensors:

```python
# For large tensors, use appropriate backend
tensor_large = np.random.rand(20, 20, 20)

# Try PyTorch backend for potential GPU acceleration
try:
    set_backend('torch')
    model_torch = EMPR(tensor_large)
    result_torch = model_torch.decompose(order=2)
    print("PyTorch backend successful")
except:
    print("PyTorch not available, using NumPy")
    set_backend('numpy')
    model_cpu = EMPR(tensor_large)
    result_cpu = model_cpu.decompose(order=2)

# For very large tensors, consider lower order approximations
model = EMPR(tensor_large)
result_fast = model.decompose(order=1)  # Faster, main effects only
result_accurate = model.decompose(order=3)  # Slower, full interactions
```

📊 Component Interpretation

Understanding HDMR/EMPR components:

Component	Description	Mathematical Form	Interpretation
`g1`, `g2`, `g3`	Main effects	$f(x_1)$, $f(x_2)$, $f(x_3)$	Individual variable impacts
`g12`, `g13`, `g23`	Two-way interactions	$f(x_1,x_2)$, $f(x_1,x_3)$, $f(x_2,x_3)$	Pairwise variable interactions
`g123`	Three-way interactions	$f(x_1,x_2,x_3)$	Complex multi-variable interactions

Decomposition Formula:

$$F(x_1,x_2,x_3) \approx f_0 + g_1(x_1) + g_2(x_2) + g_3(x_3) + g_{12}(x_1,x_2) + g_{13}(x_1,x_3) + g_{23}(x_2,x_3) + g_{123}(x_1,x_2,x_3)$$

🎯 Best Practices

1. Backend Selection

- **NumPy**: Default choice, good for most applications (CPU only)
- **PyTorch**: Better for integration with deep learning workflows (CPU/GPU)
- **TensorFlow**: Good for large-scale distributed computing (CPU/GPU)

2. Support Vector Choice

- **DAS (Data-Adaptive Supports)**: Usually provides best approximation quality
- **Ones**: Simpler, faster, good for well-behaved functions
- **Custom**: For domain-specific knowledge or special function properties

3. Order Selection

- **Order 1**: Fast, captures main effects only
- **Order 2**: Good balance of speed and accuracy for most applications
- **Order 3+**: Full accuracy but exponentially more expensive

4. Weight Configuration (HDMR)

- **Average**: Good default choice for most functions
- **Gaussian**: Better for smooth functions
- **Chebyshev**: Good for polynomial-like functions
- **Custom**: When you have domain knowledge about function behavior

---

🔧 Troubleshooting

Common Issues

1. Import Errors

```python
# If you get import errors, ensure you're in the project directory
import sys
import os
sys.path.insert(0, os.path.dirname(__file__))
```

2. Backend Not Available

```python
from backends import set_backend
```

```python
try:
    set_backend('torch')
except ValueError as e:
    print(f"Backend error: {e}")
    set_backend('numpy')  # Fallback to NumPy
```

3. Memory Issues with Large Tensors

```python
# For memory issues, reduce decomposition order or tensor size
tensor_large = np.random.rand(50, 50, 50)

# Instead of order=3, use order=2 or order=1
model = EMPR(tensor_large)
result = model.decompose(order=1)  # Uses less memory
```

4. Poor Approximation Quality

```python
# Try different support vectors or increase order
model_das = EMPR(tensor, supports='das')
model_ones = EMPR(tensor, supports='ones')

result_das = model_das.decompose(order=2)
result_ones = model_ones.decompose(order=2)

mse_das = np.mean((tensor - result_das) ** 2)
mse_ones = np.mean((tensor - result_ones) ** 2)

print(f"DAS MSE: {mse_das:.6e}, Ones MSE: {mse_ones:.6e}")
```

📈 Performance Benchmarks

Typical performance on various tensor sizes (NumPy backend, order=2):

Tensor Size	Elements	Time (seconds)	Memory (MB)
5×5×5	125	0.001	<1
10×10×10	1,000	0.01	~5
20×20×20	8,000	0.1	~50
50×50×50	125,000	2.0	~500

Performance varies with hardware and decomposition order

🤝 Contributing

We welcome contributions! To contribute:

1. Fork the repository
2. Create a feature branch: `git checkout -b feature-name`
3. Make your changes and add tests
4. Run tests: `python -m pytest tests/`
5. Submit a pull request

📝 License

📚 References

1. Sobol, I. M. (2001). Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Mathematics and Computers in Simulation*, 55(1-3), 271-280.

2. Rabitz, H., & Aliş, Ö. F. (1999). General foundations of high-dimensional model representations. *Journal of Mathematical Chemistry*, 25(2-3), 197-233.

3. Li, G., Wang, S. W., & Rabitz, H. (2002). Practical approaches to construct RS-HDMR component functions. *The Journal of Physical Chemistry A*, 106(37), 8721-8733.

📞 Support

- 📧 Email: [your-email@domain.com]
- 🐛 Issues: GitHub Issues
- 💬 Discussions: GitHub Discussions

Made with 💖 for the scientific computing community