MDLLossTorch API Documentation

Overview

MDLLossTorch is a PyTorch library that implements Minimum Description Length (MDL) loss functions for neural network training. The library computes the total "cost" of a model in bits, combining both parameter encoding costs and residual encoding costs.

Total MDL Loss = Parameter Bits + Residual Bits

Installation

```bash
pip install mdllosstorch
```

For census data integration tests:

```bash
pip install mdllosstorch[census]
```

Core API

MDLLoss

The main loss function class that combines residual and parameter encoding.

```python
from mdllosstorch import MDLLoss

loss = MDLLoss(
    method="yeo-johnson",        # Transform method: "yeo-johnson" or "box-cox"
    data_resolution=1e-6,        # Discretization resolution for residuals
    param_resolution=1e-6,       # Discretization resolution for parameters
    include_transform_param_bits=True,  # Include transform parameter encoding costs
    lam_grid=None,               # Custom lambda grid for transforms
    coder="gauss_nml",           # Residual coder: "gauss_nml" or transform-based
    use_parallel_sa=False        # Use parallel simulated annealing vs grid search
)

# Forward pass
bits = loss(original, reconstructed, model)
```

Parameters:

- `method`: Transform method for residuals (`"yeo-johnson"` or `"box-cox"`)
- `data_resolution`: Discretization resolution for data values (default: 1e-6)
- `param_resolution`: Discretization resolution for parameters (default: 1e-6)
- `include_transform_param_bits`: Whether to include transform parameter costs
- `lam_grid`: Custom grid of lambda values for transform search
- `coder`: Residual encoding method:
  - `"gauss_nml"`: Direct Gaussian encoding with quantization awareness
  - Transform-based: Uses specified `method` with Yeo-Johnson/Box-Cox
- `use_parallel_sa`: Use parallel simulated annealing instead of grid search

Returns:

- `torch.Tensor`: Total MDL loss in bits (scalar, differentiable)

Convenience Functions

compute_mdl()

Compute MDL loss with automatic data resolution estimation.

```python
from mdllosstorch import compute_mdl

bits = compute_mdl(
    x,                      # Original data tensor
    yhat,                   # Reconstructed data tensor
    model,                  # PyTorch model
    method="yeo-johnson",        # Transform method
    data_resolution="auto",      # "auto" or float value
    param_resolution=1e-6        # Parameter discretization
)
```

report_mdl()

Detailed MDL breakdown with statistics.

```python
```

```python
from mdllosstorch import report_mdl

report = report_mdl(x, yhat, model, data_resolution="auto")
print(report)
# {
#     "total_bits": 15234.56,
#     "bits_per_entry": 2.34,
#     "parameter_bits": 1234.56,
#     "residual_bits": 14000.0,
#     "data_resolution": 1e-5,
#     "method": "yeo-johnson"
# }
```

Component Functions

Residual Encoding

residual_bits_transformed_gradsafe()

Transform-based residual encoding with Yeo-Johnson or Box-Cox.

```python
from mdllosstorch import residual_bits_transformed_gradsafe

bits = residual_bits_transformed_gradsafe(
    original,                    # Original tensor
    reconstructed,               # Reconstructed tensor
    lam_grid=None,               # Lambda values to search
    method="yeo-johnson",        # Transform method
    offset_c=None,               # Box-Cox offset (auto if None)
    include_param_bits=True,     # Include transform parameter costs
    data_resolution=1e-6,        # Data discretization resolution
    use_parallel_sa=False        # Use parallel simulated annealing
)
```

residual_bits_transformed_softmin()

Fully differentiable softmin version (experimental).

```python
```

```python
from mdllosstorch import residual_bits_transformed_softmin

bits = residual_bits_transformed_softmin(
    original, reconstructed,
    lam_grid=None,
    tau=1.0,                    # Softmin temperature
    method="yeo-johnson",
    # ... other parameters same as above
)
```

Parameter Encoding

parameter_bits_student_t_gradsafe()

Student-t prior for individual parameter tensors.

python

```python
from mdllosstorch import parameter_bits_student_t_gradsafe

bits = parameter_bits_student_t_gradsafe(
    w,                          # Parameter tensor
    include_param_bits=True,    # Include hyperparameter encoding costs
    param_resolution=1e-6,      # Parameter discretization resolution
    nu_grid=(1.5, 2, 3, 5, 8, 16, 32, 64),  # Degrees of freedom to search
    sigma_scales=(0.25, 0.5, 1.0, 2.0, 4.0),  # Scale factors to search
    use_parallel_sa=False       # Use parallel simulated annealing
)
```

parameter_bits_model_student_t()

Student-t encoding for entire model.

python

```python
from mdllosstorch import parameter_bits_model_student_t

bits = parameter_bits_model_student_t(
    model,                      # PyTorch model
    include_param_bits=True,    # Include hyperparameter costs
    param_resolution=1e-6,      # Parameter discretization
    use_parallel_sa=False       # Use parallel simulated annealing
)
```

Hyperparameter Search Methods

Grid Search (Default)

Exhaustive search over predefined parameter grids.

Pros:

- Deterministic results
- Guaranteed global optimum within grid
- Simple and reliable

Cons:

- Computationally expensive (40-80 evaluations per forward pass)
- Sequential execution (doesn't utilize GPU parallelism)

Parallel Simulated Annealing

Intelligent hyperparameter search with adaptive exploration.

```python
# Enable parallel SA for both parameter and residual encoding
loss = MDLLoss(use_parallel_sa=True)
```

Pros:

- ~8x computational speedup
- ~30x better memory efficiency
- Adaptive convergence over training epochs
- GPU-friendly parallel evaluation

Cons:

- Stochastic results (small variance due to randomness)
- More complex implementation

Configuration:

- **Batch size**: Automatically adapts to available memory
- **Temperature schedule**: Adaptive cooling with periodic reheating
- **Exploration**: 30% random exploration, 70% local search around best solutions

Usage Examples

Basic Neural Network Training

```python
import torch
import torch.nn as nn
from mdllosstorch import MDLLoss

# Define model and data
model = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 784)
)

# MDL loss with automatic data resolution
loss_fn = MDLLoss(data_resolution="auto", use_parallel_sa=True)

# Training loop
for batch in dataloader:
    x = batch
    yhat = model(x)

    # MDL loss combines reconstruction quality + model complexity
    loss = loss_fn(x, yhat, model)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Autoencoder with Custom Resolution

```python
```

```python
# High-precision application
loss_fn = MDLLoss(
    data_resolution=1e-8,      # Very fine discretization
    param_resolution=1e-8,     # High parameter precision
    method="box-cox",          # Box-Cox transforms
    use_parallel_sa=True       # Fast hyperparameter search
)

autoencoder = nn.Sequential(
    nn.Linear(1000, 100),  # Encoder
    nn.Linear(100, 1000)   # Decoder
)

# Forward pass
encoded = autoencoder[0](data)
reconstructed = autoencoder[1](encoded)
loss = loss_fn(data, reconstructed, autoencoder)
```

Model Selection

```python
# Compare different architectures using MDL
models = [
    nn.Linear(100, 100),                    # Simple
    nn.Sequential(nn.Linear(100, 50), nn.Linear(50, 100)),  # Medium
    nn.Sequential(nn.Linear(100, 200), nn.Linear(200, 100)) # Complex
]

loss_fn = MDLLoss()
for i, model in enumerate(models):
    yhat = model(data)
    mdl_bits = loss_fn(data, yhat, model)
    print(f"Model {i}: {mdl_bits.item():.2f} bits")

# Lower MDL = better trade-off between fit and complexity
```

Detailed Analysis

```python
```

```
# Get detailed breakdown
report = report_mdl(data, reconstruction, model, data_resolution="auto")

print(f"Total: {report['total_bits']:.1f} bits")
print(f"Per sample: {report['bits_per_entry']:.3f} bits/sample")
print(f"Parameters: {report['parameter_bits']:.1f} bits")
print(f"Residuals: {report['residual_bits']:.1f} bits")
print(f"Auto resolution: {report['data_resolution']:.2e}")
```

Advanced Configuration

Custom Lambda Grids

```python
# Custom transform parameter search space
custom_grid = torch.linspace(-1.5, 1.5, 61)  # Narrower range, finer resolution

loss_fn = MDLLoss(
    method="yeo-johnson",
    lam_grid=custom_grid,
    use_parallel_sa=False  # Use grid search with custom grid
)
```

Memory-Constrained Environments

```python
# Limit memory usage for large models/datasets
from mdllosstorch.parallel_sa import MDLParallelHyperparameterSearch

# Custom SA with memory limit
class MemoryEfficientMDL(MDLLoss):
    def __init__(self, **kwargs):
        super().__init__(use_parallel_sa=True, **kwargs)
        # Override SA search with memory constraints
        MDLParallelHyperparameterSearch.__init__ = lambda self, n_parallel=4, memory_limit_mb=50: \
            super(MDLParallelHyperparameterSearch, self).__init__(n_parallel, memory_limit_mb)

loss_fn = MemoryEfficientMDL()
```

Performance Guidelines

When to Use Parallel SA vs Grid Search

Use Parallel SA when:

- Training on GPU
- Large datasets (>10K parameters)
- Speed is critical
- Memory is limited

Use Grid Search when:

- Reproducibility is essential
- Small models (<1K parameters)
- CPU-only training
- Research/debugging scenarios

Hyperparameter Selection

Data Resolution:

- Use `"auto"` for most applications
- Manual values: 1e-6 to 1e-3 typical range
- Too small: numerical instability
- Too large: loss of sensitivity

Parameter Resolution:

- 1e-6: Good default
- 1e-8: High precision applications
- 1e-4: Fast/approximate training

Transform Method:

- `"yeo-johnson"`: Handles positive/negative values, good default
- `"box-cox"`: Positive values only, sometimes more stable

Troubleshooting

Common Issues

NaN or Inf losses:

- Check data for extreme values
- Reduce data_resolution (try 1e-4)
- Ensure model outputs are reasonable

Slow training:

- Enable `use_parallel_sa=True`
- Reduce custom lambda grid size
- Check for very large models

Memory errors:

- Reduce batch size
- Use `coder="gauss_nml"` (more memory efficient)
- Custom memory limits in parallel SA

Debugging

```python
# Check individual components
from mdllosstorch import parameter_bits_model_student_t, residual_bits_transformed_gradsafe

param_bits = parameter_bits_model_student_t(model)
residual_bits = residual_bits_transformed_gradsafe(data, reconstruction)

print(f"Parameters: {param_bits.item():.2f} bits")
print(f"Residuals: {residual_bits.item():.2f} bits")
print(f"Ratio: {param_bits.item() / residual_bits.item():.3f}")

# Healthy ratios typically 0.01 - 1.0
# Very high parameter bits → model too complex
# Very high residual bits → poor reconstruction
```

Theory Background

The MDL principle seeks models that minimize the total description length:

L(M,D) = L(M) + L(D|M)

Where:

- **L(M)**: Cost to encode the model parameters (Student-t prior)
- **L(D|M)**: Cost to encode the data given the model (transform + Gaussian)

This naturally balances model complexity against reconstruction quality, providing principled regularization without manual hyperparameter tuning.

References

- Rissanen, J. (1978). Modeling by shortest data description.
- Box, G.E.P. and Cox, D.R. (1964). An analysis of transformations.

- Yeo, I.K. and Johnson, R.A. (2000). A new family of power transformations.