mdllosstorch

PyTorch Library for Minimum Description Length Loss Functions

API Documentation

Table of Contents

1. Overview

mdllosstorch is a PyTorch library that implements Minimum Description Length (MDL) loss functions for neural network training. The library calculates the total "cost" of a model in bits, combining both the cost of encoding the model parameters and the cost of encoding the residuals (prediction errors).

> MDL is an information-theoretic principle that balances model complexity against prediction accuracy. A model that requires fewer bits to encode both its parameters and its prediction errors is considered better according to MDL theory.

2. Core Concept

MDL Loss = Parameter Bits + Residual Bits

Parameter Bits

Cost of encoding model weights using Student-t priors with automatic hyperparameter selection

Residual Bits

Cost of encoding prediction errors using transformed Gaussian distributions (Yeo-Johnson or Box-Cox)

3. Main API Components

3.1 MDLLoss (Primary Interface)

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

```
from mdllosstorch import MDLLoss

loss_fn = MDLLoss(
    method="yeo-johnson",                 # or "box-cox"
    include_transform_param_bits=True,  # include lambda parameter costs
    data_resolution=1e-6,               # discretization resolution for data
    param_resolution=1e-6,              # discretization resolution for parameters
    lam_grid=None                       # custom lambda values to search
)

# Use like any PyTorch loss function
total_loss = loss_fn(original_data, model_predictions, model)
```

Parameters:

- `method` : Transform method ("yeo-johnson" or "box-cox")
- `include_transform_param_bits` : Include costs for transform parameters
- `data_resolution` : Discretization resolution for residual data
- `param_resolution` : Discretization resolution for model parameters
- `lam_grid` : Custom lambda values for transform search

3.2 Parameter Encoding Functions

parameter_bits_student_t_gradsafe

Calculates MDL bits for a single parameter tensor using Student-t priors.

```
from mdllosstorch import parameter_bits_student_t_gradsafe

bits = parameter_bits_student_t_gradsafe(
    w,                             # parameter tensor
    include_param_bits=True,       # include hyperparameter costs
    param_resolution=1e-6,         # discretization resolution
    nu_grid=(1.5, 2, 3, 5, 8, 16, 32, 64),    # Student-t degrees of freedom
    sigma_scales=(0.25, 0.5, 1.0, 2.0, 4.0)  # scale factors for sigma
)
```

parameter_bits_model_student_t

Calculates total MDL bits for all trainable parameters in a model.

```
from mdllosstorch import parameter_bits_model_student_t

total_param_bits = parameter_bits_model_student_t(
    model,                         # PyTorch model
    include_param_bits=True,       # include hyperparameter costs
```

```
    param_resolution=1e-6       # discretization resolution
)
```

3.3 Residual Encoding Functions

residual_bits_transformed_gradsafe (Recommended)

Stable gradient version that selects transform parameters without gradients.

```
from mdllosstorch import residual_bits_transformed_gradsafe

bits = residual_bits_transformed_gradsafe(
    original,                    # ground truth data
    reconstructed,              # model predictions
    lam_grid=None,              # lambda values to search (auto-generated if None)
    method="yeo-johnson",       # or "box-cox"
    offset_c=None,              # Box-Cox offset (auto-computed if None)
    include_param_bits=True,    # include transform parameter costs
    data_resolution=1e-6        # discretization resolution
)
```

residual_bits_transformed_softmin

Fully differentiable version using softmin over lambda grid.

```
from mdllosstorch import residual_bits_transformed_softmin

bits = residual_bits_transformed_softmin(
    original, reconstructed,
    lam_grid=None,
    tau=1.0,                    # temperature for softmin
    method="yeo-johnson",
    offset_c=None,
```

```
    include_param_bits=True,
    data_resolution=1e-6
)
```

4. Key Features

Transform Methods

- **Yeo-Johnson:** Works with any real-valued residuals
- **Box-Cox:** Requires positive residuals (adds offset automatically)

Gradient Stability

The `gradsafe` functions select transformation parameters without gradients, then apply them with gradients for stable training.

Automatic Hyperparameter Selection

Automatically searches over transformation parameters (lambda) and Student-t parameters (nu, sigma) to minimize description length.

Discretization Costs

Includes the theoretical cost of discretizing continuous values to finite precision.

🎯 Use Cases

This library is particularly useful for:

- Model selection and comparison
- Regularization with information-theoretic principles
- Balancing model complexity against prediction accuracy
- Scenarios requiring principled model complexity penalties

5. Typical Usage Patterns

Basic Training Loop

```
import torch
from mdllosstorch import MDLLoss

# Initialize loss function
mdl_loss = MDLLoss(method="yeo-johnson")

# In training loop
for epoch in range(num_epochs):
    for batch_data, batch_targets in dataloader:
        optimizer.zero_grad()

        # Forward pass
        predictions = model(batch_data)

        # Calculate MDL loss
        loss = mdl_loss(batch_targets, predictions, model)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        print(f'Epoch {epoch}, Loss: {loss.item():.4f} bits')
```

Custom Lambda Grid

```
# Use custom transformation parameter search space
import torch
from mdllosstorch import MDLLoss

custom_lambdas = torch.linspace(-3.0, 3.0, 121)
mdl_loss = MDLLoss(
```

```
    method="yeo-johnson",
    lam_grid=custom_lambdas
)
```

Analyzing Individual Components

```
from mdllosstorch import (
    parameter_bits_model_student_t,
    residual_bits_transformed_gradsafe
)

# Analyze parameter and residual costs separately
param_bits = parameter_bits_model_student_t(model)
residual_bits = residual_bits_transformed_gradsafe(targets, predictions)

print(f"Parameter bits: {param_bits:.2f}")
print(f"Residual bits: {residual_bits:.2f}")
print(f"Total MDL cost: {param_bits + residual_bits:.2f}")
```

6. Mathematical Background

Minimum Description Length Principle

The MDL principle states that the best model is the one that provides the shortest description of the data. This description length includes:

- **Model Description:** Bits needed to encode the model parameters
- **Data Description:** Bits needed to encode the data given the model

Parameter Encoding

Model parameters are encoded using Student-t distributions:

- The library searches over degrees of freedom (nu) and scale (sigma) parameters
- Uses robust median-based scale estimation
- Includes discretization costs for finite precision representation

Residual Encoding

Prediction residuals are transformed to approximate Gaussianity:

- **Yeo-Johnson Transform:** Handles both positive and negative values
- **Box-Cox Transform:** Requires positive values (offset added automatically)
- Jacobian correction ensures proper probability density transformation
- Gaussian encoding after transformation minimizes description length

⚠️ Important Notes

- The library assumes residuals follow the specified transformed distribution
- Gradient stability is maintained by selecting transform parameters off-graph
- Discretization costs account for finite precision in real implementations
- All costs are measured in bits (log base 2)

mdllosstorch API Documentation

Generated on: 8/29/2025