utils

HuangFuSL

Aug 22, 2025

CONTENTS

CHAPTER

ONE

UTILS DOCUMENTATION

1.1 Usage

First mount this repo as a submodule under the source tree of your project. For example if your project is in `src/` directory.

```
git submodule add https://github.com/HuangFuSL/utils src/utils
```

Then you can import the modules in your Python code:

```
from utils import cprint
```

Use the following script to update the submodule:

```
git submodule update --remote --merge
```

CHAPTER

TWO

PYTORCH UTILITIES

2.1 Automatic Device Selection

utils.ctorch.device - Utilities for managing and monitoring GPU devices in PyTorch.

class utils.ctorch.device.**GpuStat**(*idx*, *name*, *avg_util*, *avg_free_gb*, *total_gb*)

Bases: object

Dataclass that represents the status of a GPU device.

Parameters

- **idx** (*int*) – Index of the GPU.
- **name** (*str*) – Name of the GPU.
- **avg_util** (*float*) – Average utilization percentage over the sampling period.
- **avg_free_gb** (*float*) – Average free memory in GB over the sampling period.
- **total_gb** (*float*) – Total memory in GB of the GPU.

avg_free_gb: float

avg_util: float

idx: int

name: str

total_gb: float

utils.ctorch.device.**get_best_device**(*window_sec=3.0*, *interval_sec=0.5*)

Automatically selects the best available device based on GPU utilization and memory.

Parameters

- **window_sec** (*float*) – Total time in seconds to collect GPU statistics.
- **interval_sec** (*float*) – Time in seconds between each sample.

Returns

The best device identifier, either 'mps', 'cuda:<idx>', or 'cpu'.

Return type

str

2.2 Functional Operators

utils.ctorch.functional - Functional utilities for PyTorch tensors.

utils.ctorch.functional.**gradient_reversal**(*x*, *alpha=1.0*)

Apply a gradient reversal layer to the input tensor. The forward pass is the identity function, but during backpropagation, the gradient is multiplied by -alpha.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **alpha** (*float*) – Scaling factor for the gradient reversal. Default is 1.0.

Returns
The input tensor with the gradient reversed during backpropagation.

Return type
torch.Tensor

utils.ctorch.functional.**log_norm_pdf**(*x*, *mean*, *Sigma=None*, *logSigma=None*, *batch_first=None*)

Calculate the log probability density function of a normal distribution.

$$\log p(x) = -\frac{1}{2}\left(D\log(2\pi) + \log|\Sigma| + (x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

Parameters

- **x** (*torch.Tensor*) – Input tensor, shape (N, D), where N is the number of samples and D is the number of dimensions.
- **mean** (*torch.Tensor*) – Mean of the normal distribution, shape (D,), or (N, D)
- **Sigma** (*torch.Tensor | None*) – Covariance matrix of the normal distribution, shape (N, D, D), (N, D), (N,), (D, D), (D,), or a scalar.
- **logSigma** (*torch.Tensor | None*) – Logarithm of the covariance matrix, same shape as Sigma.
- **batch_first** (*bool | None*) – If True, indicates that the first dimension of Sigma is the batch size (N).

Returns
Tensor containing the log PDF values.

Return type
torch.Tensor

utils.ctorch.functional.**mmd_distance**(*x*, *y*, ** (Keyword-only parameters separator (PEP 3102))*, *sigma=None*, *gamma=None*, *reduce=False*)

Compute the Maximum Mean Discrepancy (MMD) distance between two sets of tensors.

The MMD distance is given by:

$$\text{MMD}(x,y) = K(x,x) - 2K(x,y) + K(y,y)$$

Parameters

- **x** (*torch.Tensor*) – First tensor, shape (N, D), where N is the number of samples and D is the number of features.
- **y** (*torch.Tensor*) – Second tensor, shape (M, D), where M is the number of samples and D is the number of features.

- **sigma** (*torch.Tensor | int | float | None*) – Bandwidth parameter for the RBF kernel, scalar, or shape (K,), where K is the number of kernels.
- **gamma** (*torch.Tensor | int | float | None*) – 1 / (2 * sigma^2) parameter for the RBF kernel, scalar, or shape (K,), where K is the number of kernels.
- **reduce** (*bool | torch.Tensor*) – Whether to reduce the output. * If True, returns the mean MMD distance under different bandwidths. * If False, returns the MMD distance for each bandwidth. * If a tensor, it should have shape (K,) and will be used as mean weight.

Returns
Tensor containing the MMD distance values, shape (K,) if reduce is False, or a scalar otherwise,

Return type
torch.Tensor

utils.ctorch.functional.**norm_pdf**(*x*, *mean*, *Sigma=None*, *logSigma=None*)

Calculate the log probability density function of a normal distribution.

Parameters

- **x** (*torch.Tensor*) – Input tensor, shape (N, D), where N is the number of samples and D is the number of dimensions.
- **mean** (*torch.Tensor*) – Mean of the normal distribution, shape (D,), or (N, D)
- **Sigma** (*torch.Tensor | None*) – Covariance matrix of the normal distribution, shape (N, D, D), (N, D), (N,), (D, D), (D,), or a scalar.
- **logSigma** (*torch.Tensor | None*) – Logarithm of the covariance matrix, same shape as Sigma.
- **batch_first** (*bool*) – If True, indicates that the first dimension of Sigma is the batch size (N).

Returns
Tensor containing the PDF values.

Return type
torch.Tensor

utils.ctorch.functional.**rbf_kernel**(*x*, *y=None*, ***, *sigma=None*, *gamma=None*, *reduce=False*)

Compute the Radial Basis Function (RBF) kernel between two sets of tensors.

The RBF kernel is given by:

$$K(x, y) = \exp(-\gamma||x - y||^2)$$

or equivalently,

$$K(x, y) = \exp\left(-\frac{||x - y||^2}{2\sigma^2}\right)$$

Parameters

- **x** (*torch.Tensor*) – First tensor, shape (N, D), where N is the number of samples and D is the number of features.
- **y** (*torch.Tensor | None*) – Second tensor, shape (M, D), where M is the number of samples and D is the number of features.
- **sigma** (*torch.Tensor | int | float | None*) – Bandwidth parameter for the RBF kernel, scalar, or shape (K,), where K is the number of kernels.

- **gamma** (*torch.Tensor | int | float | None*) – 1 / (2 * sigma^2) parameter for the RBF kernel, scalar, or shape (K,), where K is the number of kernels.
- **reduce** (*torch.Tensor | bool*) – Whether to reduce the output.
  - If True, returns the mean of RBF kernel values under different bandwidths.
  - If False, returns the RBF kernel values for each bandwidth.
  - If a tensor, it should have shape (K,) and will be used as mean weight.

Returns
Tensor containing the RBF kernel values, shape (N, M) or (K, N, M) if multiple kernels are used.

Return type
torch.Tensor

utils.ctorch.functional.**wasserstein_distance**(*x*, *y*, *p=2.0*, *eps=1e-06*, *wasser_iters=20*, *wasser_eps=0.001*)

Compute the Wasserstein distance between two sets of tensors using the Sinkhorn algorithm.

The Wasserstein distance is given by:

$$W_p(x, y) = \left(\inf_{\gamma \in \Gamma(x,y)} \int ||x - y||^p d\gamma(x, y) \right)^{1/p}$$

Parameters

- **x** (*torch.Tensor*) – First tensor, shape (N, D), where N is the number of samples and D is the number of features.
- **y** (*torch.Tensor*) – Second tensor, shape (M, D), where M is the number of samples and D is the number of features.
- **p** (*float*) – Order of the norm to use for the distance calculation.
- **eps** (*float*) – Small value to avoid division by zero.
- **wasser_iters** (*int*) – Number of iterations for the Sinkhorn algorithm.
- **wasser_eps** (*float*) – Epsilon value for the Sinkhorn algorithm.

Returns
Tensor containing the Wasserstein distance value.

Return type
torch.Tensor

2.3 Additional Metrics

utils.ctorch.metrics - Utility functions for computing metrics in PyTorch

class utils.ctorch.metrics.**BatchedAUC**(*nbins=1000*, *device='cpu'*, *logit=False*)

Bases: *BatchedMetric*

Batched AUC metric for binary classification tasks. This class computes the AUC in a batch-wise manner using a generator.

```
auc_metric = BatchedAUC(nbins=1000, device='cpu', logit=False)
for batch in data_loader:
    y_true, y_score = batch
```

(continues on next page)

(continued from previous page)

```
    auc_metric(y_true, y_score)
auc_value = auc_metric.finalize()
```

accumulator(*nbins=1000*, *device='cpu'*, *logit=False*)

Compute the AUC in a batch-wise manner using a generator.

Parameters

- **nbins** (*int*) – Number of bins to use for the histogram.
- **device** (*str | torch.device*) – Device to use for computation.
- **logit** (*bool*) – If True, apply sigmoid to the scores before computing AUC.

class utils.ctorch.metrics.**BatchedHitRate**(*k=None*)

Bases: *BatchedMetric*

accumulator(*k=None*)

Compute the hit rate in a batch-wise manner using a generator.

Parameters

k (*int*) – The number of top predictions to consider.

class utils.ctorch.metrics.**BatchedMetric**(***kwargs*)

Bases: ABC

abstractmethod accumulator()

Create a generator that accumulates metric values from batches of (y_true, y_score).

Return type

Generator[None, Optional[Tuple[Tensor, Tensor]], float]

Returns

Generator that yields None to receive batches and returns the computed metric value.

finalize()

Finalize the metric computation and return the accumulated value.

Returns

The computed metric value.

Return type

float

reset()

Reset the metric accumulator to start a new computation.

class utils.ctorch.metrics.**BatchedNDCG**(*k=None*)

Bases: *BatchedMetric*

accumulator(*k=None*)

Compute the NDCG in a batch-wise manner using a generator.

Parameters

k (*int*) – The number of top predictions to consider.

class utils.ctorch.metrics.**MetricFormatter**(*name*, *starting_epoch=0*, *larger_better=True*, *eps=0.0005*)

Bases: object

Formats the output of metrics during training.

format()

Format the metric for display.

Returns
Formatted string representation of the metric.

Return type
str

update(*value*)

Update the metric with a new value.

Parameters
value (*float*) – The new value to update the metric with.

utils.ctorch.metrics.**auc_score**(*y_true*, *y_score*)

Compute the Area Under the Curve (AUC) for binary classification.

Parameters

- **y_true** (*torch.Tensor*) – Ground truth binary labels (0 or 1).
- **y_score** (*torch.Tensor*) – Predicted scores or probabilities.

Returns
The computed AUC value.

Return type
float

utils.ctorch.metrics.**auuc_score**(*y*, *t*, *tau*, *qini=True*, *normalize=True*)

Compute the Area Under the Uplift Curve (AUUC).

Parameters

- **y** (*torch.Tensor*) – Binary outcome (0 or 1).
- **t** (*torch.Tensor*) – Binary treatment assignment (0 or 1).
- **tau** (*torch.Tensor*) – Uplift scores.
- **qini** (*bool*) – If True, compute Qini instead of AUUC (subtracts the random baseline).

Returns
The computed AUUC value.

Return type
float

utils.ctorch.metrics.**hit_rate**(*y_true*, *y_score*, *k=None*)

Compute the hit rate for retrieval tasks.

Parameters

- **y_true** (*torch.Tensor*) – Shape (N, C), ground truth binary labels (0 or 1).
- **y_score** (*torch.Tensor*) – Shape (N, C), predicted scores or probabilities.
- **k** (*int*) – The number of top predictions to consider.

Returns
The computed hit rate.

Return type
float

utils.ctorch.metrics.**ndcg_score**(*y_true*, *y_score*, *k=None*)

Compute the normalized discounted cumulative gain (NDCG) for retrieval tasks.

Parameters

- **y_true** (*torch.Tensor*) – Shape (N, C), ground truth binary labels (0 or 1).
- **y_score** (*torch.Tensor*) – Shape (N, C), predicted scores or probabilities.
- **k** (*int*) – The number of top predictions to consider.

Returns
The computed NDCG value.

Return type
float

2.4 Additional Neural Network Layers

nn.py - Utilities Modules for PyTorch tensors

Originally in ctorch.py

class utils.ctorch.nn.**Activation**(*name*, **args*, ***kwargs*)

Bases: *Module*

Arbitrary activation function module.

Parameters

- **name** (*str*) – The name of the activation function.
- ***args** – Positional arguments for the activation function.
- ****kwargs** – Keyword arguments for the activation function.

forward(*x*)

Forward pass for the activation function.

Return type
Tensor

class utils.ctorch.nn.**DNN**(**layer_dims*, *layer_type=<class 'torch.nn.modules.linear.Linear'>*, *flip_gradient=False*, *batchnorm=False*, *bias=True*, *dropout=None*, *activation='relu'*, *residual=False*)

Bases: *Module*

A Deep Neural Network (DNN) or Multi-Layer Perceptron (MLP) module.

Parameters

- **layer_dims** (**int*) – The dimensions of each layer in the network, including input and output dimensions.
- **layer_type** (*Type[torch.nn.Module]*) – The type of layer to use (e.g., Linear).
- **flip_gradient** (*bool*) – Whether to apply a gradient reversal layer at the beginning.
- **batchnorm** (*bool*) – Whether to apply batch normalization after each linear layer.
- **bias** (*bool*) – Whether to include a bias term in the linear layers.
- **dropout** (*float | None*) – Dropout rate to apply after each layer. If None, no dropout is applied.

- **activation** (*str | None*) – Activation function to apply after each layer.
- **residual** (*bool*) – Whether to add a residual connection from input to output, requiring input and output dimensions to match.

Shapes:

- Input shape: (*, layer_dims[0])
- Output shape: (*, layer_dims[-1])

forward(*x*)

Forward pass for the DNN module.

Parameters
x (*torch.Tensor*) – Input tensor of shape (*, layer_dims[0]).

Returns
Output tensor of shape (*, layer_dims[-1]).

Return type
torch.Tensor

class utils.ctorch.nn.**DeEmbedding**(*embedding*)

Bases: *Module*

forward(*x*)

Forward pass for the de-embedding layer.

Parameters
x (*torch.Tensor*) – Tensor of shape (*, D), where D is the embedding dimension.

Returns
Tensor of shape (*, num_embeddings), where num_embeddings is the size of the embedding.

Return type
torch.Tensor

class utils.ctorch.nn.**FeatureEmbedding**(*num_features*, *embedding_size*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*)

Bases: *Module*

An embedding layer for encoding N multiple categorical features.

Parameters

- **num_features** (*List[int]*) – The number of unique values for each categorical feature.
- **embedding_size** (*List[int] | int*) – The size of the embedding for each feature. If a single integer is provided, it will be used for all features.
- **padding_idx** (*int | None*)
- **max_norm** (*float | None*)
- **norm_type** (*float*)
- **scale_grad_by_freq** (*bool*)
- **sparse** (*bool*)

Shapes:

- Input shape: (*, num_features)

- Output shape: (*, sum(embedding_size))

forward(*x*)

Forward pass for the embedding layer.

Parameters
x (Tensor) – Tensor of shape (*, num_features)

Returns
Tensor of shape (*, sum(embedding_size))

Return type
torch.Tensor

property total_embedding_size: int

Gets the total embedding size, which is the sum of all individual embedding sizes.

Returns
Total embedding size.

Return type
int

class utils.ctorch.nn.**GradientReversalLayer**(*alpha=1.0*)

Bases: *Module*

A layer that reverses the gradient during backpropagation.

Parameters
alpha (*float*) – The scaling factor for the gradient reversal. Default is 1.0.

forward(*x*)

Forward pass for the gradient reversal layer.

Parameters
x (*torch.Tensor*) – Input tensor.

Return type
Tensor

class utils.ctorch.nn.**Module**(**args*, ***kwargs*)

Bases: Module

A base class for all modules in ctorch. Supports device tracking and parameter counting.

property device

Get the device of the module.

Returns
The device on which the module's parameters are located.

Return type
torch.device

property num_parameters

Get the number of parameters in the module.

Returns
The total number of parameters in the module.

Return type
int

class utils.ctorch.nn.**MonotonicLinear**(*in_features*, *out_features*, *bias=True*, *non_neg_func='softplus'*)

Bases: *Module*

Implements a monotonic linear layer. The monotonicity is enforced by applying a non-negative activation function to the weights.

Parameters

- **in_features** (*int*) – Number of input features.
- **out_features** (*int*) – Number of output features.
- **bias** (*bool*) – Whether to include a bias term.
- **non_neg_func** (*str | Callable*) – Element-wise non-negative activation function to use. Should be one of:
  - `relu`: $f(x) = \max(0, x)$
  - `softplus`: $f(x) = \log(1 + \exp(x))$
  - `sigmoid`: $f(x) = \frac{1}{1+\exp(-x)}$
  - `elu`: $f(x) = ELU(x) + 1$
  - `abs`: $f(x) = |x|$
  - `square`: $f(x) = x^2$
  - `exp`: $f(x) = e^x$

To keep the monotonicity, non-monotonic activations including `softmax`, `GELU`, `SiLU` and `Mish` should not be used. Normalization techniques such as layer normalization or batch normalization should also be avoided.

Shapes:

- Input shape: (*, in_features)
- Output shape: (*, out_features)

forward(*x*)

Forward pass for the monotonic linear layer.

Parameters
x (*torch.Tensor*) – Input tensor of shape (*, in_features).

Returns
Output tensor of shape (*, out_features).

Return type
torch.Tensor

class utils.ctorch.nn.**RotaryTemporalEmbedding**(*embedding_dim*, *denom=10000.0*)

Bases: *Module*

Implements rotary positional embedding proposed in "RoFormer: Enhanced Transformer with Rotary Position Embedding" (https://arxiv.org/abs/2104.09864).

$$\boldsymbol{R} = \text{diag}(\boldsymbol{R}_1, \ldots, \boldsymbol{R}_{\lfloor n/2 \rfloor})$$
$$\boldsymbol{R}_i = \begin{bmatrix} \cos(t\theta_i) & -\sin(t\theta_i) \\ \sin(t\theta_i) & \cos(t\theta_i) \end{bmatrix}$$
$$\theta_i = \frac{1}{10000^{2(i-1)/d_{model}}}$$

Parameters

- **embedding_dim** (*int*) – The dimension of the embedding space. Must be even.
- **denom** (*float*) – The denominator (10000.0) for the positional encoding.

Shapes:

- Input shape: x (*, embedding_dim), t (*)
- Output shape: (*, embedding_dim)

forward(*t*, *x*)

Forward pass for the rotary temporal embedding.

Parameters

- **t** (*torch.Tensor*) – Time record tensor of shape (*).
- **x** (*torch.Tensor*) – Input tensor of shape (*, embedding_dim).

Returns
Embedding of shape (*, embedding_dim).

Return type
torch.Tensor

class utils.ctorch.nn.**SinusoidalTemporalEmbedding**(*embedding_dim*, *denom=10000.0*)

Bases: *Module*

Implements sinusoidal positional embedding proposed in "Attention is All You Need".

$$PE_{(batch,pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{2k/d_{model}}}\right) & \text{if } i = 2k \\ \cos\left(\frac{pos}{10000^{2k/d_{model}}}\right) & \text{if } i = 2k+1 \end{cases}$$

Parameters

- **embedding_dim** (*int*) – The dimension of the embedding space.
- **denom** (*float*) – The denominator (10000.0) for the positional encoding.

Shapes:

- Input shape: (*, embedding_dim)
- Output shape: (*, embedding_dim)

forward(*t*)

Forward pass for the circular temporal embedding.

Parameters
t (*torch.Tensor*) – Time record tensor of shape (*).

Returns
Embedding of shape (*, embedding_dim).

Return type
torch.Tensor

class utils.ctorch.nn.**TransformerDecoderLayer**(*d_model*, *nhead*, *dim_feedforward=2048*, *dropout=0.1*, *activation=<function relu>*, *layer_norm_eps=1e-05*, *batch_first=False*, *norm_first=False*, *bias=True*, *device=None*, *dtype=None*)

Bases: TransformerDecoderLayer

get_cross_attention_map(*tgt*, *memory*, *tgt_mask=None*, *memory_mask=None*, *tgt_key_padding_mask=None*, *memory_key_padding_mask=None*, *tgt_is_causal=False*, *memory_is_causal=False*)

Get the cross-attention map from the decoder layer.

Parameters

- **tgt** (*torch.Tensor*)
- **memory** (*torch.Tensor*)
- **tgt_mask** (*torch.Tensor | None*)
- **memory_mask** (*torch.Tensor | None*)
- **tgt_key_padding_mask** (*torch.Tensor | None*)
- **memory_key_padding_mask** (*torch.Tensor | None*)
- **tgt_is_causal** (*bool*)
- **memory_is_causal** (*bool*)

Returns

The attention weights of shape (batch_size, num_heads, tgt_seq_len, memory_seq_len).

Return type

torch.Tensor

get_self_attention_map(*tgt*, *memory*, *tgt_mask=None*, *memory_mask=None*, *tgt_key_padding_mask=None*, *memory_key_padding_mask=None*, *tgt_is_causal=False*, *memory_is_causal=False*)

Get the self-attention map from the decoder layer.

Parameters

- **tgt** (*torch.Tensor*)
- **tgt_mask** (*torch.Tensor | None*)
- **memory_mask** (*torch.Tensor | None*)
- **tgt_key_padding_mask** (*torch.Tensor | None*)
- **memory_key_padding_mask** (*torch.Tensor | None*)
- **tgt_is_causal** (*bool*)
- **memory_is_causal** (*bool*)

Returns

The attention weights of shape (batch_size, num_heads, seq_len, seq_len).

Return type

torch.Tensor

class utils.ctorch.nn.**TransformerEncoderLayer**(*d_model*, *nhead*, *dim_feedforward=2048*, *dropout=0.1*, *activation=<function relu>*, *layer_norm_eps=1e-05*, *batch_first=False*, *norm_first=False*, *bias=True*, *device=None*, *dtype=None*)

Bases: TransformerEncoderLayer

A Transformer encoder layer with additional functionality to get attention maps.

get_attention_map(*src*, *src_mask=None*, *src_key_padding_mask=None*, *is_causal=False*)

Get the attention map from the encoder layer.

Parameters

- **src** (*torch.Tensor*)
- **src_mask** (*torch.Tensor | None*)
- **src_key_padding_mask** (*torch.Tensor | None*)
- **is_causal** (*bool*)

Returns
The attention weights of shape (batch_size, num_heads, seq_len, seq_len).

Return type
torch.Tensor

2.5 Additional Base Operators

utils.ctorch.ops - Utilities Operators for PyTorch tensors

class utils.ctorch.ops.**GradientReversalOp**(**args*, ***kwargs*)

Bases: Function

Gradient reversal operation for adversarial training.

static backward(*ctx*, *grad_output*)

Backward pass for the gradient reversal operation.

Parameters
grad_output (*torch.Tensor*) – Gradient from the next layer.

Returns
The gradient with the reversal applied.

Return type
torch.Tensor

static forward(*ctx*, *x*, *alpha*)

Forward pass for the gradient reversal operation.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **alpha** (*float*) – Scaling factor for the gradient reversal.

Returns
The input tensor with the gradient reversal applied.

Return type
torch.Tensor

2.6 Utilities for Variable Length Sequences

padding.py - Utilities for handling PackedSequences

Originally in ctorch.py Author: HuangFuSL Date: 2025-06-26

utils.ctorch.padding.**get_key_padding_mask_left**(*lengths*)

Create a key padding mask for sequences based on their lengths, with sequences left-aligned.

Parameters
lengths (*torch.Tensor*) – A 1D tensor containing the lengths of each sequence.

Returns
A boolean mask tensor indicating the padding positions.

Return type
torch.Tensor

utils.ctorch.padding.**get_key_padding_mask_right**(*lengths*)

Create a key padding mask for sequences based on their lengths, with sequences right-aligned.

Parameters
lengths (*torch.Tensor*) – A 1D tensor containing the lengths of each sequence.

Returns
A boolean mask tensor indicating the padding positions.

Return type
torch.Tensor

utils.ctorch.padding.**get_model_memory_size**(*model*)

Get the total memory size of a model in bytes.

Parameters
model (*torch.nn.Module*) – The input model.

Returns
The total memory size of the model in bytes.

Return type
int

utils.ctorch.padding.**get_tensor_memory_size**(*tensor*)

Get the memory size of a tensor in bytes.

Parameters
tensor (*torch.Tensor*) – The input tensor.

Returns
The memory size of the tensor in bytes.

Return type
int

utils.ctorch.padding.**pack_padded_sequence_right**(*input*, *lengths*, *batch_first=False*, *enforce_sorted=False*)

Like *torch.nn.utils.rnn.pack_padded_sequence* but accepts right-aligned sequences.

Parameters

- **input** (*torch.Tensor*) – The input tensor, which should be right-aligned.
- **lengths** (*torch.Tensor*) – A 1D tensor containing the lengths of each sequence.
- **batch_first** (*bool*) – If True, the input is expected to be of shape (batch_size, seq_len, ...). Otherwise, the input is expected to be of shape (seq_len, batch_size, ...).
- **enforce_sorted** (*bool*) – If True, the input sequences must be sorted by length in descending order. If False, the input sequences can be in any order.

Returns
A packed sequence object containing the right-aligned sequences.

Return type
torch.nn.utils.rnn.PackedSequence

utils.ctorch.padding.**packed_binary_op**(*op*, *a*, *b*)

Apply a binary element-wise operation to a PackedSequence or a regular tensor.

Parameters

- **op** (*Callable*) – A binary operation to apply.
- **a** (*PackedSequence | torch.Tensor*) – The first input data, either a PackedSequence or a regular tensor.
- **b** (*PackedSequence | torch.Tensor*) – The second input data, either a PackedSequence or a regular tensor.

Returns
The output after applying the operation.

Return type
PackedSequence | torch.Tensor

utils.ctorch.padding.**packed_concat**(*packed_seq*, *dim=-1*)

Concatenate a list of PackedSequence objects along a specified dimension. Notice that the length of the packed sequences must be the same.

Parameters

- **packed_seq** (*List[PackedSequence]*) – List of PackedSequence objects to concatenate.
- **dim** (*int*) – Dimension along which to concatenate. Default is -1 (last dimension). The dimension must not be 0 (the packed time dimension). The sequence length dimension (dimension 1 of the padded tensor where batch_size is True) is omitted.

Returns
A new PackedSequence object containing the concatenated data.

Return type
PackedSequence

utils.ctorch.padding.**packed_forward**(*module*, *packed_input*)

Forward pass for a module with packed input.

Parameters

- **module** (*torch.nn.Module*) – The neural network to apply.
- **packed_input** (*PackedSequence | torch.Tensor*) – The packed input data.

Returns
The output after applying the module. If the input is a PackedSequence, the output will also be a PackedSequence, otherwise it will be a regular tensor.

Return type
PackedSequence | torch.Tensor

utils.ctorch.padding.**packed_unary_op**(*func*, *x*)

Apply an unary element-wise function to a PackedSequence or a regular tensor.

Parameters

- **func** (*Callable*) – An element-wise function to apply.
- **x** (*PackedSequence | torch.Tensor*) – The input data, either a PackedSequence or a regular tensor.

Returns
The output after applying the function. If the input is a PackedSequence, the output will also be a PackedSequence, otherwise it will be a regular tensor.

Return type
PackedSequence | torch.Tensor

utils.ctorch.padding.**pad_packed_sequence_right**(*sequence*, *batch_first=False*, *padding_value=0.0*, *total_length=None*)

Like *torch.nn.utils.rnn.pad_packed_sequence* but right-aligns the sequences.

Parameters

- **sequence** (*torch.nn.utils.rnn.PackedSequence*) – The packed sequence to pad.
- **batch_first** (*bool*) – If True, the output will be of shape (batch_size, seq_len, ...). If False, the output will be of shape (seq_len, batch_size, ...).
- **padding_value** (*float*) – The value to use for padding.
- **total_length** (*int | None*) – If specified, the output will be padded to this length. If None, the output will be padded to the maximum length of the sequences in the packed sequence

Returns
The padded tensor and the lengths of the original sequences.

Return type
Tuple[torch.Tensor, torch.Tensor]

utils.ctorch.padding.**unpad_sequence_right**(*input*, *lengths*, *batch_first=False*)

Like *torch.nn.utils.rnn.unpad_sequence* but accepts right-aligned sequences.

Parameters

- **input** (*torch.Tensor*) – The input tensor, which should be right-aligned.
- **lengths** (*torch.Tensor*) – A 1D tensor containing the lengths of each sequence.
- **batch_first** (*bool*) – If True, the input is expected to be of shape (batch_size, seq_len, ...). Otherwise, the input is expected to be of shape (seq_len, batch_size, ...).

Returns
A list of tensors, each representing a sequence with padding removed.

Return type
List[torch.Tensor]

CHAPTER

THREE

SAMPLER UTILITIES

utils.sampler.**clean_target_name**(*name*)

CHAPTER

FOUR

LOGGING UTILITIES

utils.clogging - Colored Logging Formatter

A Python module that provides a colored logging formatter for the *logging* module. This module defines a base colored formatter class and a default implementation that applies specific colors to different logging levels and fields. It allows for easy customization of log message styles, making it easier to distinguish between different log levels in console output.

class utils.clogging.**BaseColoredFormatter**(*fmt='%(asctime)s %(levelname)-8s %(name)s: %(message)s'*, *datefmt='%Y-%m-%d %H:%M:%S'*, *style='%'*)

Bases: Formatter

Add color to log messages based on their level.

This formatter allows customization of log message styles based on their logging level. It supports coloring the level name and optionally the message itself. The styles can be customized through the *get_color* method, which should return a argument dictionary compatible with *cprint.cprint*.

Parameters

- **fmt** (*str*) – The format string for the log messages.
- **datefmt** (*str | None*) – The format string for the date in log messages.
- **style** (*str*) – The character used in the format string (default is '%').

(same as logging.Formatter)

property field_names: List[str]

Returns the list of field names that can be colored. This can be overridden in subclasses to specify which fields are available.

'base' is a special field that can be used to apply a default color to the template.

Returns
A list of field names that can be colored.

Return type
List[str]

format(*record*)

Formats the log record with colors applied to the specified fields.

Parameters
record (*logging.LogRecord*) – The log record to format.

Returns
The formatted log message with colors applied.

Return type
str

`formatTime`(*record*, *datefmt=None*)

Formats the time of the log record. If datefmt is provided, it formats the time accordingly. Otherwise, it uses the default format.

Parameters

- **record** (*logging.LogRecord*) – The log record to format.
- **datefmt** (*str | None*) – The format string for the date.

Returns
The formatted time string with color applied.

Return type
str

`get_color`(*field_name*, *level*)

Returns the color format for a given field name and logging level. This method should be overridden in subclasses to provide specific color styles for different fields and levels.

Parameters

- **field_name** (*str*) – The name of the field to color.
- **level** (*int*) – The logging level for which to get the color.

Returns
A dictionary of color attributes compatible with *utils.cprint.cprint*.

Return type
Dict[str, Any] | None

class utils.clogging.**DefaultColoredFormatter**(*fmt='%(asctime)s %(levelname)-8s %(name)s: %(message)s'*, *datefmt='%Y-%m-%d %H:%M:%S'*, *style='%'*)

Bases: *BaseColoredFormatter*

Default colored formatter with predefined styles for log levels.

`get_color`(*field_name*, *level*)

Returns the color format for a given field name and logging level. This method should be overridden in subclasses to provide specific color styles for different fields and levels.

Parameters

- **field_name** (*str*) – The name of the field to color.
- **level** (*int*) – The logging level for which to get the color.

Returns
A dictionary of color attributes compatible with *utils.cprint.cprint*.

Return type
Dict[str, Any] | None

CHAPTER

FIVE

PRINTING UTILITIES

utils.cprint - Colorful Terminal Output

A Python module for colorful terminal output with support for ANSI escape codes, xterm-256color, and true color (24-bit RGB). It provides functions to format text with foreground and background colors, styles (bold, italic, underline, strikethrough), and fallback to lower color grades if the terminal does not support the requested color depth.

utils.cprint.**cformat**(*text*, ***, *fg=None*, *bg=None*, *fallback=True*, *bf=False*, *dim=False*, *it=False*, *us=False*, *st=False*, *reset='\\x1b[0m'*)

Format a string with ANSI escape codes for colors and styles.

Parameters

- **text** (*str*) – The text to format.
- **fg** (*str | int | Tuple[int, int, int] | None*) – Foreground color (color name, xterm-256color index, RGB tuple, or hex code).
- **bg** (*str | int | Tuple[int, int, int] | None*) – Background color (same format as fg).
- **fallback** (*bool*) – Whether to use fallback colors if the terminal does not support the requested color depth.
- **bf** (*bool*) – Whether to use bold text.
- **it** (*bool*) – Whether to use italic text.
- **us** (*bool*) – Whether to underline the text.
- **st** (*bool*) – Whether to use strikethrough text (not supported in all terminals).
- **reset** (*str*) – String to append after the formatted text (default is ANSI reset code).

Returns

A formatted string with ANSI escape codes.

Return type

str

utils.cprint.**cprefix**(*fg=None*, *bg=None*, *fallback=True*, *bf=False*, *dim=False*, *it=False*, *us=False*, *st=False*)

Generate an ANSI escape code prefix for formatting text with colors and styles.

Parameters

- **fg** (*str | int | Tuple[int, int, int] | None*) – Foreground color (color name, xterm-256color index, RGB tuple, or hex code).
- **bg** (*str | int | Tuple[int, int, int] | None*) – Background color (same format as fg).

- **fallback** (*bool*) – Whether to use fallback colors if the terminal does not support the requested color depth.
- **bf** (*bool*) – Whether to use bold text.
- **it** (*bool*) – Whether to use italic text.
- **us** (*bool*) – Whether to underline the text.
- **st** (*bool*) – Whether to use strikethrough text (not supported in all terminals).

Returns
A string containing the ANSI escape code prefix.

Return type
str

utils.cprint.**cprint**(**obj*, *fg=None*, *bg=None*, *fallback=True*, *bf=False*, *dim=False*, *it=False*, *us=False*, *st=False*, *reset='\\x1b[0m'*, *sep=' '*, *end='\\n'*, *file=None*, *flush=False*)

Colorful print function with support for foreground and background colors, styles (bold, italic, underline, strikethrough), and fallback to lower color grades if the terminal does not support the requested color depth.

Parameters

- ***obj** (*Any*) – Objects to print.
- **fg** (*str | int | Tuple[int, int, int] | None*) – Foreground color (color name, xterm-256color index, RGB tuple, or hex code).
- **bg** (*str | int | Tuple[int, int, int] | None*) – Background color (same format as fg).
- **fallback** (*bool*) – Whether to use fallback colors if the terminal does not support the requested color depth.
- **bf** (*bool*) – Whether to use bold text.
- **it** (*bool*) – Whether to use italic text.
- **us** (*bool*) – Whether to underline the text.
- **st** (*bool*) – Whether to use strikethrough text (not supported in all terminals).
- **reset** (*str*) – String to append after the formatted text (default is ANSI reset code).

The following parameters are inherited from the built-in print function:

- sep: Separator between objects.
- end: String appended after the last object.
- file: A file-like object (default is sys.stdout).
- flush: Whether to forcibly flush the output buffer.

CHAPTER

SIX

ARGUMENT PARSING UTILITIES

utils.parser

A utility to automatically parse command line arguments into a dataclass instance. This module provides a decorator `@auto_cli` that can be applied to a dataclass. After applying the decorator, you can call `parse_args()` on the dataclass to parse command line arguments and return an instance of the dataclass.

1. A special field `--config` is added to allow loading configuration from a file in JSON, YAML, or TOML format.
2. When a field is specified in both the command line arguments and the configuration file, the command line argument takes precedence.

`utils.parser.`**`auto_cli`**(*cls=None*, */ (Positional-only parameter separator (PEP 570))*, ***decorator_kw*)

Automatically generates an argument parser for a dataclass with type hints. Field types, including basic types, and `Optional` are automatically recognized and converted corresponding argument types. Complex types like lists and dictionaries can be input as strings in json format.

Apart from the fields of the dataclass, `auto_cli` also adds a `--config` argument to the parser for loading configuration settings from a json, yaml, or toml file. Values from the configuration file have a lower priority than command line arguments.

The decorated function will get the following new class methods:

- **`get_parser(prefix: str = '') -> argparse.ArgumentParser:`**
  Returns an *argparse.ArgumentParser* instance with arguments based on the dataclass fields. The *prefix* argument is prepended to each argument name. The parser returned does not include the `--config` argument for loading configuration files.

  Example:

  ```
  @auto_cli
  @dataclasses.dataclass
  class YourClass:
      name: str = 'DefaultName'
      age: int = 25

  parser = YourClass.get_parser()
  parser_prefix = YourClass.get_parser(prefix='man')
  ```

  The generated *parser* will accept the following arguments:

  - `--name`: The name argument with a default value of 'DefaultName'.
  - `--age`: The age argument with a default value of 25.

  The generated `parser_prefix` will accept the following arguments:

  - `--man-name`: The name argument with a default value of 'DefaultName'.

- `--man-age`: The age argument with a default value of 25.

- **`parse_namespace(ns: argparse.Namespace, kw: Dict[str, Any] = None, prefix: str = '') -> 'YourClass':`**
  Parses an `argparse.Namespace` instance and a kwargs dictionary into an instance of the dataclass. Prefix is prepended to each argument name. The `kw` dictionary is loaded from a config file if specified in `ns`. A `ValueError` is raised if a required argument is missing from both `ns` and `kw`, and no default value is provided.

- **`parse_args(argv: List[str] = None) -> 'YourClass':`**
  Parses command line arguments and returns an instance of the dataclass. If `argv` is `None`, it uses `sys.argv[1:]`. The method also handles config files specified in the command line arguments.

Returns
The decorated class.

Return type
Type[cls]

`utils.parser.`**`get_all_parser`**(*dataclass=None*, ***dataclasses*)

Returns a combined `argparse.ArgumentParser` that merges the parsers of multiple dataclasses into one, and then adds a `--config` argument for loading configuration files.

The dataclass provided as a positional argument is treated as unprefixed, while the others are prefixed with their keyword argument names.

Parameters

- **`dataclass`** (*`Optional[Any]`*) – A single dataclass to include in the parser.
- **`**dataclasses`** (*`Any`*) – Additional dataclasses to include in the parser.

Returns
An argument parser that includes all specified dataclasses.

Return type
argparse.ArgumentParser

`utils.parser.`**`parse_all_args`**(*cli_args=None*, *dataclass=None*, ***dataclasses*)

Parse command line arguments into a dictionary of dataclass instances. Each dataclass is identified by its name in the *dataclasses* argument.

The `parse_all_args` function accepts two types of input:

1. `parse_all_args(cli_args, dataclass, **dataclasses)`: where `cli_args` is a list of command line arguments, `dataclass` is the unprefixed dataclass, and `dataclasses` are additional prefixed dataclasses.
2. `parse_all_args(dataclass, **dataclasses)`: where `dataclass` is the unprefixed dataclass and `dataclasses` are additional prefixed dataclasses. `sys.argv[1:]` is used as the command line arguments.

The result is a dictionary where the keys are the prefixes of the dataclasses and the values are instances of those dataclasses, parsed from the command line arguments. The unprefixed dataclass is stored under the key `''`.

Example:

```
@auto_cli
@dataclasses.dataclass
class ClassMain:
    name: str = 'MainName'
    age: int = 30
```

(continues on next page)

(continued from previous page)

```
@auto_cli
@dataclasses.dataclass
class ClassAdditional:
    address: str = 'DefaultAddress'
    phone: str = '1234567890'

parser = get_all_parser(ClassMain, additional=ClassAdditional)
```

The generated parser will accept the following arguments:

- `--name`: The name argument with a default value of `'MainName'`.
- `--age`: The age argument with a default value of `30`.
- `--additional-address`: The address argument with a default value of `'DefaultAddress'`.
- `--additional-phone`: The phone argument with a default value of `'1234567890'`.

Or the following configuration files:

```
name: 'MainName'
age: 30
additional_address: 'DefaultAddress'
additional_phone: '1234567890'
```

Parameters

- **cli_args** (*List[str] | Any | None*) – Command line arguments to parse. If None, uses `sys.argv[1:]`.
- **dataclass** (*Any | None*) – A single dataclass to include in the parsing.
- ****dataclasses** (*Any*) – Additional dataclasses to include in the parsing.

Returns

A dictionary where keys are dataclass names and values are instances of those dataclasses.

Return type

Dict[str, Any]

PYTHON MODULE INDEX

u

INDEX

W