
utils

HuangFuSL

Oct 10, 2025

CONTENTS

1	utils documentation	1
1.1	Usage	1
2	PyTorch Utilities	3
2.1	Automatic Device Selection	3
2.2	Functional Operators	4
2.3	Additional Metrics	7
2.4	Additional Neural Network Layers	9
2.5	Additional Base Operators	19
2.6	Utilities for Variable Length Sequences	20
2.7	Utilities for Reinforcement Learning	25
3	Sampler Utilities	37
4	Logging Utilities	39
5	Printing Utilities	41
6	Argument Parsing Utilities	43
	Python Module Index	47
	Index	49

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
```


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 back-propagation, 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.logit_product(x, y)`

Calculate the logit of the product of two probabilities given their logits.

$$\sigma^{-1}(\sigma(x) \cdot \sigma(y)) = x + y - \log(1 + e^x + e^y)$$

Parameters

- **x** (*torch.Tensor*) – Logit of the first probability.
- **y** (*torch.Tensor*) – Logit of the second probability.

Returns

Logit of the product of the two probabilities.

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*) – $1 / (2 * \text{sigma}^2)$ 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*) – $1 / (2 * \text{sigma}^2)$ 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
    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.CholeskyTrillinear(in_features, out_dim, bias=True, eps=0.0001, scale=None, non_neg_func='softplus')`

Bases: [Module](#)

Implements a linear layer that returns a lower-triangular matrix that is positive definite.

Parameters

- **in_features** (*int*) – Number of input features.
- **out_dim** (*int*) – The output matrix dimension.
- **bias** (*bool*) – Whether to include a bias term.
- **eps** (*float*) – The small value added to the main diagonal of the matrix
- **scale** (*float*) – The maximum scale of the matrix elements.
- **non_neg_func** (*str* | *Callable*) – Element-wise non-negative activation function on the diagonal elements.
 - **softplus**: $f(x) = \log(1 + \exp(x))$
 - **elu**: $f(x) = ELU(x) + 1$
 - **sigmoid**: $f(x) = 1/(1 + e^{-x})$
 - **square**: $f(x) = x^2$
 - **exp**: $f(x) = e^x$

Shapes:

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

diag(x)

Return the main diagonal of the resulting matrix.

Parameters

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

Return type

Tensor

Shapes:

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

forward(x)

Forward pass for the positive definite linear layer.

Parameters

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

Returns

Tril tensor of shape (*, out_dim, out_dim).

Return type

torch.Tensor

pd(x)

Return the positive definite LL^\top .

Parameters

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

Return type

Tensor

Shapes:

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

```
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 | [Activation](#) | 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.**FactorizedNoisyLinear**(*in_features*, *out_features*, *bias=True*, *init_sigma=0.5*)Bases: [Module](#)Implements a noisy linear layer according to <https://arxiv.org/abs/1706.10295>

The layer works the same way as a standard linear layer, but with added noise during training.

$$\begin{aligned}z_{\text{in}} &\sim \mathcal{N}(0, I_{d_{\text{in}}}) \\z_{\text{out}} &\sim \mathcal{N}(0, I_{d_{\text{out}}}) \\f(x) &= \text{sign}(x) \odot \sqrt{|x|} \\w &= w_{\mu} + w_{\sigma} \odot (f(z_{\text{in}})f(z_{\text{out}})^{\top}) \\b &= b_{\mu} + b_{\sigma} \odot f(z_{\text{out}})\end{aligned}$$

The parameters are initialized as:

$$\begin{aligned}w_{\mu}, b_{\mu} &\sim \mathcal{U}(-1/\sqrt{d_{\text{in}}}, 1/\sqrt{d_{\text{int}}}) \\w_{\sigma}, b_{\sigma} &= \sigma_{\text{init}}/\sqrt{d_{\text{in}}}\end{aligned}$$

Parameters

- **in_features** (*int*) – Number of input features.
- **out_features** (*int*) – Number of output features.
- **bias** (*bool*) – Whether to include a bias term.
- **init_sigma** (*float*) – (float): The initial sigma coefficient σ_{init} , default is 0.5.

Shapes:

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

static **f**(*x*)**Return type**

Tensor

forward(*x*)

Forward pass for the noisy 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.**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.GaussianLinear(in_features, out_features, bias=True, mu_scale=None,
                                   Sigma_scale=None, eps=0.0001, non_neg_func='softplus',
                                   gaussian_type=<class
                                   'torch.distributions.multivariate_normal.MultivariateNormal'>)
```

Bases: [Module](#)

Implements a linear layer that returns a multivariate Gaussian distribution

Parameters

- **in_features** (*int*) – Number of input features.
- **out_features** (*int*) – Number of output features.
- **bias** (*bool*) – Whether to include a bias term.
- **mu_scale** (*float | None*) – If not None, the mean value is scaled by a tanh function with the given scale.
- **cov_scale** (*float*) – The boundary value for covariance matrix elements.
- **eps** (*float*) – Minimum value added to the diagonal of the covariance matrix.

- **non_neg_func** (*str* / *Callable*) – Non-negative mapping of the diagonal elements of the covariance matrix.
- **gaussian_type** ([MultivariateNormalClass](#)) – The constructor of target distribution.

Shapes:

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

cov(*x*)

Return the covariance matrix of the target distribution.

Parameters

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

Return type

Tensor

Shapes:

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

forward(*x*)

Forward pass for the positive definite linear layer.

Parameters

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

Returns

The target distribution.

Return type

torch.distributions.Distribution

mean(*x*)

Return the mean value of the target distribution.

Parameters

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

Return type

Tensor

Shapes:

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

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.IndependentNoisyLinear`(*in_features*, *out_features*, *bias=True*, *init_sigma=0.017*)Bases: *Module*Implements a noisy linear layer according to <https://arxiv.org/abs/1706.10295>

The layer works the same way as a standard linear layer, but with added noise during training.

$$w = w_{\mu} + w_{\sigma} \odot \varepsilon$$

$$b = b_{\mu} + b_{\sigma} \odot \varepsilon$$

The parameters are initialized as:

$$w_{\mu}, b_{\mu} \sim \mathcal{U}(-1/\sqrt{d_{\text{in}}}, 1/\sqrt{d_{\text{int}}})$$

$$w_{\sigma}, b_{\sigma} = \sigma_{\text{init}}$$

Parameters

- **in_features** (*int*) – Number of input features.
- **out_features** (*int*) – Number of output features.
- **bias** (*bool*) – Whether to include a bias term.
- **init_sigma** (*float*) – (*float*): The initial sigma coefficient σ_{init} , default is 0.017.

Shapes:

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

forward(*x*)

Forward pass for the noisy 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.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: $(*, \text{in_features})$
- Output shape: $(*, \text{out_features})$

forward(x)

Forward pass for the monotonic linear layer.

Parameters

x (*torch.Tensor*) – Input tensor of shape $(*, \text{in_features})$.

Returns

Output tensor of shape $(*, \text{out_features})$.

Return type

`torch.Tensor`

class `utils.ctorch.nn.MultivariateNormalClass`(*args, **kwargs)

Bases: `Protocol`

Internal protocol for `GaussianLinear` module. Any valid `gaussian_type` parameter should follow the following protocol.

```
def __call__(
    self, loc: torch.Tensor, covariance_matrix: torch.Tensor | None = None,
    precision_matrix: torch.Tensor | None = None,
    scale_tril: torch.Tensor | None = None,
    validate_args: Any = None
) -> torch.distributions.Distribution:
    ...
```

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>).

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

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”.

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

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.append_right(input_tensor, append_value=0.0, num_steps=1, batch_first=True)`

Add `num_steps` steps at the end of the time axis, to a packed sequence or tensor.

Parameters

- **input_tensor** (*PackedOrTensor*) – The input tensor or packed sequence to pad.
- **padding_value** (*float, optional*) – The value to use for padding. Defaults to 0.0.
- **num_steps** (*int, optional*) – The number of steps to add. Defaults to 1.
- **batch_first** (*bool, optional*) – If True, the input tensor is assumed to be in (batch, seq, ...) format, otherwise (seq, batch, ...), only effective when accepting tensors. Defaults to True.

Returns

The padded tensor or packed sequence.

Return type

PackedOrTensor

`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.masked_select(values_input, mask_input)`

Perform masked selection of variable length on a PackedSequence or a regular tensor.

Parameters

- **values_input** (*PackedOrTensor*) – The input values to select from.
- **mask_input** (*PackedOrTensor*) – The mask indicating which values to select.

Returns

A PackedSequence containing only the selected values.

Return type

PackedSequence

Example

```

from torch.nn.utils.rnn import pad_packed_sequence
values_list = [
    [[1., 10.], [2., 20.], [3., 30.]],
    [[4., 40.], [5., 50.], [0., 0.]],
]
mask_list = [
    [True, False, True],
    [False, True, False],
]

values = torch.tensor(values_list, dtype=torch.float32)
mask = torch.tensor(mask_list, dtype=torch.bool)

out = masked_select(values, mask)
out_padded, out_len = pad_packed_sequence(out, batch_first=True)

```

(continues on next page)

(continued from previous page)

```

assert out_padded.tolist() == [
    [[1.0, 10.0], [3.0, 30.0]],
    [[5.0, 50.0], [0.0, 0.0]],
]
assert out_len.tolist() == [2, 1]

```

`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 sample-wise function to a PackedSequence or a regular tensor.

Parameters

- **func** (*Callable*) – An sample-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.prepend_left(input_tensor, append_value=0.0, num_steps=1, batch_first=True)`

Add `num_steps` steps at the beginning of the time axis, to a packed sequence or tensor.

Parameters

- **input_tensor** (*PackedOrTensor*) – The input tensor or packed sequence to pad.
- **padding_value** (*float, optional*) – The value to use for padding. Defaults to 0.0.
- **num_steps** (*int, optional*) – The number of steps to add. Defaults to 1.
- **batch_first** (*bool, optional*) – If True, the input tensor is assumed to be in (batch, seq, ...) format, otherwise (seq, batch, ...), only effective when accepting tensors. Defaults to True.

Returns

The padded tensor or packed sequence.

Return type

PackedOrTensor

`utils.ctorch.padding.truncate_left(input_tensor, num_steps=1, batch_first=True, strict=True)`

Remove `num_steps` steps at the beginning of the time axis, from a packed sequence or tensor.

Parameters

- **input_tensor** (*PackedOrTensor*) – The input tensor or packed sequence to truncate.
- **num_steps** (*int, optional*) – The number of steps to remove. Defaults to 1.
- **batch_first** (*bool, optional*) – If True, the input tensor is assumed to be in (batch, seq, ...) format, otherwise (seq, batch, ...), only effective when accepting tensors. Defaults to True.
- **strict** (*bool, optional*) – If True, raises an error if truncating action will yield an empty sequence.

Returns

The truncated tensor or packed sequence.

Return type

PackedOrTensor

`utils.ctorch.padding.truncate_right(input_tensor, num_steps=1, batch_first=True, strict=True)`

Remove `num_steps` steps at the end of the time axis, from a packed sequence or tensor.

Parameters

- **input_tensor** (*PackedOrTensor*) – The input tensor or packed sequence to truncate.
- **num_steps** (*int, optional*) – The number of steps to remove. Defaults to 1.
- **batch_first** (*bool, optional*) – If True, the input tensor is assumed to be in (batch, seq, ...) format, otherwise (seq, batch, ...), only effective when accepting tensors. Defaults to True.
- **strict** (*bool, optional*) – If True, raises an error if truncating action will yield an empty sequence.

Returns

The truncated tensor or packed sequence.

Return type

PackedOrTensor

`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]

2.7 Utilities for Reinforcement Learning

rl.py - Utilities Components for Reinforcement Learning

class `utils.ctorch.rl.BasePolicyNetwork(state_shape, *, gamma=0.99, tau=1)`

Bases: [`BaseRLModel`](#), [`TargetNetworkMixin`](#)

Abstract base class for policy-based reinforcement learning models.

Parameters

- **state_shape** (`Tuple[int, ...]`) – The shape dimensions of the input state.
- **gamma** (`float, optional`) – The discount factor for future rewards, defaults to 0.99.
- **tau** (`int, optional`) – The number of steps to look ahead for target updates, defaults to 1.

act(`state`)

Return the action sampled from the policy given the state.

Parameters

state (`torch.Tensor`) – The input state tensor.

Returns

A tuple containing the sampled action and its log probability.

Return type

Tuple[torch.Tensor, torch.Tensor]

cumulative_reward(`r, lambda_=1`)

Calculate the cumulative reward for a trajectory, or the GAE(lambda) estimate.

Parameters

- **r** (`torch.Tensor`) – The reward tensor of shape (L,), for GAE, r is the state-value TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$.
- **lambda_** (`float, optional`) – The lambda parameter for GAE. Defaults to 1 for cumulative reward.

Returns

The cumulative reward or GAE estimate of shape (L,).

Return type`torch.Tensor`

Shapes:

- `r`: (L,)
- `output`: (L,)

abstractmethod forward(*state*)

Forward pass to compute the action distribution given the state.

Parameters**state** (`torch.Tensor`) – The input state tensor.**Returns**

The action distribution given the state.

Return type`torch.distributions.Distribution`**log_pi(*state, action*)**

Return the log probability of the action given the state.

Return type`Tensor`**normalize_trajectory(*r, eps=1e-08*)****Return type**`Tensor`**pi(*state, action*)**

Return the action distribution probability given the state.

Return type`Tensor`**policy_parameters()**

Get the parameters of the policy network.

Returns

An iterator over the parameters of the policy network.

Return type`Iterator[torch.nn.Parameter]`**setup_target()**

Create a target network by copying the current network. This method can only be called once.

property value_model: [BaseValueNetwork](#)Get the state value sub-network. If not implemented, raises `NotImplementedError`.**value_parameters()**

Get the parameters of the value network.

Returns

An iterator over the parameters of the value network.

Return type`Iterator[torch.nn.Parameter]`

class `utils.ctorch.rl.BaseQNetwork`(*state_shape*, *num_actions*, *, *gamma*=0.99, *tau*=1)

Bases: `BaseValueNetwork`, `ABC`

Abstract base class for DQN with discrete action space.

One should either implement the `forward` method to return Q-values for all actions or specified actions.

Parameters

- **state_shape** (*Tuple[int, ...]*) – The shape dimensions of the input state.
- **num_actions** (*int*) – The number of actions the agent can take.
- **gamma** (*float, optional*) – The discount factor for future rewards, defaults to 0.99.
- **tau** (*int, optional*) – The number of steps to look ahead for target updates, defaults to 1.

A_all(*state*)

Compute the advantage function $A(s, a)$ for a given state s and action a .

$$A(s, a) = Q(s, a) - V(s)$$

Parameters

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

Returns

The advantage function $A(s, a)$ for a given state s and all actions.

Return type

`torch.Tensor`

Shapes:

- state: (*, (state_dims,))
- action: (*)
- output: (*)

Q_all(*state*)

Compute the action-value function $Q(s, a)$ for all actions a given state s .

Parameters

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

Returns

The action-value function $Q(s, a)$ for all actions a .

Return type

`torch.Tensor`

Shapes:

- state: (*, (state_dims,))
- output: (*, num_actions)

V(*state*)

Compute the state-value function $V(s)$ for a given state s .

Parameters

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

Returns

The state-value function $V(s)$ for a given state s .

Return type

`torch.Tensor`

Shapes:

- state: $(*, (state_dims,))$
- output: $(*)$

act(state, eps=0.0, sample_wise=True)

Policy for selecting actions based on the current state and exploration rate. By default, the policy takes exploration actions with a probability of ε and exploitation actions with a probability of $1 - \varepsilon$.

Parameters

state (`torch.Tensor`) – The input state tensor.

Returns

The selected actions.

Return type

`torch.Tensor`

exploit(state)

The exploitation policy, selecting actions based on the current Q-values.

Parameters

state (`torch.Tensor`) – The input state tensor.

Returns

The selected actions.

Return type

`torch.Tensor`

explore(state)

The exploration policy, defaulting to random actions.

Parameters

state (`torch.Tensor`) – The input state tensor.

Returns

The selected actions.

Return type

`torch.Tensor`

abstractmethod forward(state, action=None)

Returns the Q-values for the given state and action.

Parameters

- **state** (`torch.Tensor`) – The input state tensor.
- **action** (`torch.Tensor` | `None`) – The input action tensor.

Returns

The Q-values for the given state and action. If action is None, returns the Q-value over all actions.

Return type

torch.Tensor

td_step(*trajectory*, *a_prime=None*)

The loss function for training the value network using the TD loss. If *a_prime* is provided, the action-value function $Q(s, a)$ is used. Otherwise, the state-value function $V(s)$ is used.

$$\begin{aligned} \text{LHS} &= V(s) \\ \text{RHS} &= \begin{cases} r + V(s') & \text{if not terminal} \\ r & \text{if terminal} \end{cases} \end{aligned}$$

The LHS and RHS can be optimized using `MSELoss` or `SmoothL1Loss`.

Parameters

- **trajectory** ([Trajectory](#)) – A batch of trajectories.
- **a_prime** (*torch.Tensor* / *None*) – The next action tensor. If provided, the action-value function $Q(s, a)$ is used. If not provided, the state-value function $V(s)$ is used.

Returns

The LHS and RHS of the TD loss.

Return type

Tuple[torch.Tensor, torch.Tensor]

class `utils.ctorch.rl.BaseRLModel`(*state_shape*, *, *gamma=0.99*, *tau=1*)

Bases: [Module](#), [ABC](#)

Abstract base class for reinforcement learning models.

Parameters

- **state_shape** (*Tuple[int, ...]*) – The shape dimensions of the input state.
- **gamma** (*float*, *optional*) – The discount factor for future rewards, defaults to 0.99.
- **tau** (*int*, *optional*) – The number of steps to look ahead for target updates, defaults to 1.

abstractmethod `act`(*state*, ***kwargs*)

Policy for selecting actions based on the current state and exploration rate. By default, the policy takes exploration actions with a probability of ϵ and exploitation actions with a probability of $1 - \epsilon$.

Parameters

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

Returns

A tuple containing the selected action and the log probability of the action.

Return type

Tuple[torch.Tensor, torch.Tensor]

abstractmethod `loss`()

Return type

Tensor

class `utils.ctorch.rl.BaseValueNetwork`(*state_shape*, *, *gamma=0.99*, *tau=1*)

Bases: [BaseRLModel](#), [TargetNetworkMixin](#)

Abstract base class for Value Networks.

One should either implement the `forward` method, or `V` method.

Parameters

- **state_shape** (*Tuple[int, ...]*) – The shape dimensions of the input state.
- **gamma** (*float, optional*) – The discount factor for future rewards, defaults to 0.99.
- **tau** (*int, optional*) – The number of steps to look ahead for target updates, defaults to 1.

A(*state, action*)Compute the advantage function $A(s, a)$ for a given state s and action a .

$$A(s, a) = Q(s, a) - V(s)$$

Parameters

- **state** (*torch.Tensor*) – The input state tensor.
- **action** (*torch.Tensor*) – The input action tensor.

ReturnsThe advantage function $A(s, a)$ for a given state s and action a .**Return type***torch.Tensor***Shapes:**

- state: (*, (state_dims,))
- action: (*)
- output: (*)

Q(*state, action*)Compute the action-value function $Q(s, a)$ for a given state s and action a .**Parameters**

- **state** (*torch.Tensor*) – The input state tensor.
- **action** (*torch.Tensor*) – The input action tensor.

ReturnsThe action-value function $Q(s, a)$ for a given state s and action a .**Return type***torch.Tensor***Shapes:**

- state: (*, (state_dims,))
- action: (*, (action_dims,)) or (*)
- output: (*)

V(*state*)Compute the state-value function $V(s)$ for a given state s .**Parameters**

- **state** (*torch.Tensor*) – The input state tensor.

Returns

The state-value function $V(s)$ for a given state s .

Return type

`torch.Tensor`

Shapes:

- state: $(*, (state_dims,))$
- output: $(*)$

act(*state*, ***kwargs*)

Policy for selecting actions based on the current state and exploration rate. By default, the policy takes exploration actions with a probability of ε and exploitation actions with a probability of $1 - \varepsilon$.

Parameters

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

Returns

A tuple containing the selected action and the log probability of the action.

Return type

`Tuple[torch.Tensor, torch.Tensor]`

abstractmethod forward(*state*, *action=None*)

Forward pass of the value network. The implementation should either implement the state-value function $V(s)$ or the action-value function $Q(s, a)$.

Parameters

- **state** (*torch.Tensor*) – The input state tensor.
- **action** (*torch.Tensor* / *None*) – The input action tensor. If provided, the network should compute the action-value function $Q(s, a)$. If not provided, the network should compute the state-value function $V(s)$.

Returns

The value function $V(s)$ for the given state.

Return type

`torch.Tensor`

Shapes:

- state: $(*, (state_dims,))$
- action: $(*, (action_dims,))$, $(*)$, or *None*
- output: $(*)$

td_step(*trajectory*, *a_prime=None*)

The loss function for training the value network using the TD loss. If *a_prime* is provided, the action-value function $Q(s, a)$ is used. Otherwise, the state-value function $V(s)$ is used.

$$\begin{aligned} \text{LHS} &= V(s) \\ \text{RHS} &= \begin{cases} r + V(s') & \text{if not terminal} \\ r & \text{if terminal} \end{cases} \end{aligned}$$

The LHS and RHS can be optimized using `MSELoss` or `SmoothL1Loss`.

Parameters

- **trajectory** ([Trajectory](#)) – A batch of trajectories.
- **a_prime** (*torch.Tensor* / *None*) – The next action tensor. If provided, the action-value function $Q(s, a)$ is used. If not provided, the state-value function $V(s)$ is used.

Returns

The LHS and RHS of the TD loss.

Return type

Tuple[torch.Tensor, torch.Tensor]

class `utils.ctorch.rl.CircularTensor`(*size*, *dtype=torch.float32*)

Bases: [Module](#)

Implements a circular buffer for storing tensors. Supports `state_dict()` and `load_state_dict()` for checkpointing.

Parameters

- **size** (*int*) – The maximum number of elements in the buffer.
- **dtype** (*torch.dtype*, *optional*) – The data type of the elements in the buffer. Defaults to `torch.float32`.

append(*value*)

Append a batch of new elements to the buffer.

Parameters

value (*Any*) – The new elements to append. Supports any input that can be converted to a tensor.

Shapes:

- Input shape: (B, *)

as_numpy(*force=False*)

Get the underlying numpy array.

as_tensor()

Get the underlying tensor.

forward(*x*)

Get a batch of elements according to the index.

property size: `int`

Get the maximum size of the buffer.

Returns

The maximum size of the buffer.

Return type

`int`

class `utils.ctorch.rl.PrioritizedReplayBuffer`(*size*, *continuous_action=False*, *p_max=1000.0*)

Bases: [ReplayBuffer](#)

Implements a prioritized replay buffer for storing and sampling experiences.

Parameters

- **size** (*int*) – The maximum size of the buffer.

- **continuous_action** (*bool*, *optional*) – Whether the action space is continuous. Defaults to False.
- **p_max** (*float* | *int*, *optional*) – Default priority value for newly incoming experiences. Defaults to 1e3.

get_ipw(*idx*, *beta*, *eps*=0.0005)

Get the inverse probability weighting (IPW) for a batch of experiences, to balance the loss function. The IPW weight is given by:

$$\tilde{w}_i = \left(\frac{N \cdot w_i}{\sum_j w_j} \right)^\beta$$

Parameters

- **idx** (*torch.Tensor*) – The indices of the experiences to retrieve.
- **beta** (*float*) – The beta parameter for the importance sampling.

Returns

The importance sampling weights for the specified experiences.

Return type

torch.Tensor

get_weight(*idx*)

Get the sampling weight for a batch of experiences.

Return type

Tensor

sample_index(*batch_size*)

Sample a batch of indices from the replay buffer.

Return type

Tensor

set_weight(*idx*, *weight*)

Set the sampling weight for a batch of experiences.

store(*trajectory*)

Store a batch of new experience in the buffer.

class `utils.ctorch.rl.ReplayBuffer`(*size*, *continuous_action*=False)

Bases: *Module*

Implements a replay buffer for storing and sampling experiences.

Parameters

- **size** (*int*) – The maximum size of the buffer.
- **continuous_action** (*bool*, *optional*) – Whether the action space is continuous. Defaults to False.

forward(*idx*)

Sample a batch of experiences from the buffer.

Return type

Trajectory

get_batch(*idx*)

Sample a batch of experiences from the buffer.

Return type

Trajectory

property length

Get the current length of the buffer.

sample_index(*batch_size*)

Randomly sample a batch of indices from the buffer.

Return type

Tensor

store(*trajectory*)

Store a batch of new experience in the buffer.

class `utils.ctorch.rl.TargetNetworkMixin`

Bases: *Module*

Mixin class for models with a target network.

setup_target()

Create a target network by copying the current network. This method can only be called once.

property target: **Self**

The target network for the Q-learning algorithm. Used in double DQN. If not set, the current network is used.

Returns

The target network or self if not set.

Return type

Module

update_target(*weight=1.0*)

Update the target network by copying the weights from the current network. No-op if the target network is not used.

Parameters

weight (*float*, *optional*) – The interpolation weight for the update. By default, it is 1.0.

class `utils.ctorch.rl.Trajectory`(*state, action, reward, next_state, done, log_pi, total_reward*)

Bases: object

A batch of states, actions, and rewards collected from the environment.

Parameters

- **state** (*torch.Tensor*) – The states in the trajectory of shape (L, *state_shape).
- **action** (*torch.Tensor*) – The actions taken in the trajectory of shape (L, *action_shape).
- **reward** (*torch.Tensor*) – The rewards received in the trajectory of shape (L,).
- **next_state** (*torch.Tensor*) – The next states in the trajectory of shape (L, *state_shape).
- **done** (*torch.Tensor*) – The done flags in the trajectory of shape (L,).
- **log_pi** (*torch.Tensor*) – The log probabilities of the actions taken in the trajectory of shape (L,).

- **total_reward** (*float*) – The total reward of the trajectory. *nan* if the batch is not a trajectory.

action: Tensor

clone()

done: Tensor

get(*key*)

Return type

Tensor

log_pi: Tensor

next_state: Tensor

reward: Tensor

state: Tensor

tensor_fields: ClassVar[List[str]] = ['state', 'action', 'reward', 'next_state', 'done', 'log_pi']

total_reward: float

`utils.ctorch.rl.run_episode(env, model, max_episode_steps=None, reward_shape=<function _default_shape>, **act_kwargs)`

Simulates a single episode in the environment using the given model.

Parameters

- **env** (*gymnasium.Env*) – The environment to simulate.
- **model** (*BaseRLModel*) – The RL model to use for action selection.
- **eps** (*float*) – The exploration rate.
- **max_episode_steps** (*int* | *None*) – The maximum number of steps in the episode.
- **reward_shape** (*Callable[[s, a, r, s_prime, terminated, truncated], torch.Tensor]*) – A function to adjust the reward based on the current reward, arrived state, terminal state, and time limit truncation. Returns the adjusted reward to be passed to the model.

Returns

The collected trajectory.

Return type

Trajectory

`utils.ctorch.rl.torch_step(env, device='cpu')`

A wrapper for the environment step function to convert actions from torch tensors and results to torch tensors.

Parameters

env (*gymnasium.Env*) – The environment to wrap.

Returns

The wrapped env.step function.

Return type

Callable[[torch.Tensor], Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, Dict[str, Any]]]

SAMPLER UTILITIES

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

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.

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 unprefix, 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 unprefix dataclass, and `dataclasses` are additional prefixed dataclasses.
2. `parse_all_args(dataclass, **dataclasses)`: where `dataclass` is the unprefix 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 unprefix 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

- `utils, ??`
- `utils.clogging, 39`
- `utils.cprint, 41`
- `utils.ctorch, 3`
- `utils.ctorch.device, 3`
- `utils.ctorch.functional, 4`
- `utils.ctorch.metrics, 7`
- `utils.ctorch.nn, 9`
- `utils.ctorch.ops, 19`
- `utils.ctorch.padding, 20`
- `utils.ctorch.rl, 25`
- `utils.parser, 43`
- `utils.sampler, 37`

A

[A\(\)](#) (*utils.ctorch.rl.BaseValueNetwork method*), 30
[A_all\(\)](#) (*utils.ctorch.rl.BaseQNetwork method*), 27
[accumulator\(\)](#) (*utils.ctorch.metrics.BatchedAUC method*), 7
[accumulator\(\)](#) (*utils.ctorch.metrics.BatchedHitRate method*), 7
[accumulator\(\)](#) (*utils.ctorch.metrics.BatchedMetric method*), 7
[accumulator\(\)](#) (*utils.ctorch.metrics.BatchedNDCG method*), 8
[act\(\)](#) (*utils.ctorch.rl.BasePolicyNetwork method*), 25
[act\(\)](#) (*utils.ctorch.rl.BaseQNetwork method*), 28
[act\(\)](#) (*utils.ctorch.rl.BaseRLModel method*), 29
[act\(\)](#) (*utils.ctorch.rl.BaseValueNetwork method*), 31
[action](#) (*utils.ctorch.rl.Trajectory attribute*), 35
[Activation](#) (*class in utils.ctorch.nn*), 9
[append\(\)](#) (*utils.ctorch.rl.CircularTensor method*), 32
[append_right\(\)](#) (*in module utils.ctorch.padding*), 20
[as_numpy\(\)](#) (*utils.ctorch.rl.CircularTensor method*), 32
[as_tensor\(\)](#) (*utils.ctorch.rl.CircularTensor method*), 32
[auc_score\(\)](#) (*in module utils.ctorch.metrics*), 8
[auto_cli\(\)](#) (*in module utils.parser*), 43
[auuc_score\(\)](#) (*in module utils.ctorch.metrics*), 8
[avg_free_gb](#) (*utils.ctorch.device.GpuStat attribute*), 3
[avg_util](#) (*utils.ctorch.device.GpuStat attribute*), 3

B

[backward\(\)](#) (*utils.ctorch.ops.GradientReversalOp static method*), 19
[BaseColoredFormatter](#) (*class in utils.clogging*), 39
[BasePolicyNetwork](#) (*class in utils.ctorch.rl*), 25
[BaseQNetwork](#) (*class in utils.ctorch.rl*), 26
[BaseRLModel](#) (*class in utils.ctorch.rl*), 29
[BaseValueNetwork](#) (*class in utils.ctorch.rl*), 29
[BatchedAUC](#) (*class in utils.ctorch.metrics*), 7
[BatchedHitRate](#) (*class in utils.ctorch.metrics*), 7
[BatchedMetric](#) (*class in utils.ctorch.metrics*), 7
[BatchedNDCG](#) (*class in utils.ctorch.metrics*), 7

C

[cformat\(\)](#) (*in module utils.cprint*), 41

[CholeskyTrilLinear](#) (*class in utils.ctorch.nn*), 9
[CircularTensor](#) (*class in utils.ctorch.rl*), 32
[clone\(\)](#) (*utils.ctorch.rl.Trajectory method*), 35
[cov\(\)](#) (*utils.ctorch.nn.GaussianLinear method*), 14
[cprefix\(\)](#) (*in module utils.cprint*), 41
[cprint\(\)](#) (*in module utils.cprint*), 42
[cumulative_reward\(\)](#) (*utils.ctorch.rl.BasePolicyNetwork method*), 25

D

[DeEmbedding](#) (*class in utils.ctorch.nn*), 11
[DefaultColoredFormatter](#) (*class in utils.clogging*), 40
[device](#) (*utils.ctorch.nn.Module property*), 15
[diag\(\)](#) (*utils.ctorch.nn.CholeskyTrilLinear method*), 10
[DNN](#) (*class in utils.ctorch.nn*), 11
[done](#) (*utils.ctorch.rl.Trajectory attribute*), 35

E

[exploit\(\)](#) (*utils.ctorch.rl.BaseQNetwork method*), 28
[explore\(\)](#) (*utils.ctorch.rl.BaseQNetwork method*), 28

F

[f\(\)](#) (*utils.ctorch.nn.FactorizedNoisyLinear static method*), 12
[FactorizedNoisyLinear](#) (*class in utils.ctorch.nn*), 12
[FeatureEmbedding](#) (*class in utils.ctorch.nn*), 12
[field_names](#) (*utils.clogging.BaseColoredFormatter property*), 39
[finalize\(\)](#) (*utils.ctorch.metrics.BatchedMetric method*), 7
[format\(\)](#) (*utils.clogging.BaseColoredFormatter method*), 39
[format\(\)](#) (*utils.ctorch.metrics.MetricFormatter method*), 8
[formatTime\(\)](#) (*utils.clogging.BaseColoredFormatter method*), 40
[forward\(\)](#) (*utils.ctorch.nn.Activation method*), 9
[forward\(\)](#) (*utils.ctorch.nn.CholeskyTrilLinear method*), 10
[forward\(\)](#) (*utils.ctorch.nn.DeEmbedding method*), 11
[forward\(\)](#) (*utils.ctorch.nn.DNN method*), 11

- `forward()` (*utils.ctorch.nn.FactorizedNoisyLinear method*), 12
`forward()` (*utils.ctorch.nn.FeatureEmbedding method*), 13
`forward()` (*utils.ctorch.nn.GaussianLinear method*), 14
`forward()` (*utils.ctorch.nn.GradientReversalLayer method*), 14
`forward()` (*utils.ctorch.nn.IndependentNoisyLinear method*), 15
`forward()` (*utils.ctorch.nn.MonotonicLinear method*), 16
`forward()` (*utils.ctorch.nn.RotaryTemporalEmbedding method*), 17
`forward()` (*utils.ctorch.nn.SinusoidalTemporalEmbedding method*), 18
`forward()` (*utils.ctorch.ops.GradientReversalOp static method*), 19
`forward()` (*utils.ctorch.rl.BasePolicyNetwork method*), 26
`forward()` (*utils.ctorch.rl.BaseQNetwork method*), 28
`forward()` (*utils.ctorch.rl.BaseValueNetwork method*), 31
`forward()` (*utils.ctorch.rl.CircularTensor method*), 32
`forward()` (*utils.ctorch.rl.ReplayBuffer method*), 33
- ## G
- `GaussianLinear` (*class in utils.ctorch.nn*), 13
`get()` (*utils.ctorch.rl.Trajectory method*), 35
`get_all_parser()` (*in module utils.parser*), 44
`get_attention_map()`
 (*utils.ctorch.nn.TransformerEncoderLayer method*), 19
`get_batch()` (*utils.ctorch.rl.ReplayBuffer method*), 33
`get_best_device()` (*in module utils.ctorch.device*), 3
`get_color()` (*utils.clogging.BaseColoredFormatter method*), 40
`get_color()` (*utils.clogging.DefaultColoredFormatter method*), 40
`get_cross_attention_map()`
 (*utils.ctorch.nn.TransformerDecoderLayer method*), 18
`get_ipw()` (*utils.ctorch.rl.PrioritizedReplayBuffer method*), 33
`get_key_padding_mask_left()` (*in module utils.ctorch.padding*), 20
`get_key_padding_mask_right()` (*in module utils.ctorch.padding*), 20
`get_model_memory_size()` (*in module utils.ctorch.padding*), 21
`get_self_attention_map()`
 (*utils.ctorch.nn.TransformerDecoderLayer method*), 18
`get_tensor_memory_size()` (*in module utils.ctorch.padding*), 21
- ## H
- `hit_rate()` (*in module utils.ctorch.metrics*), 8
- ## I
- `idx` (*utils.ctorch.device.GpuStat attribute*), 3
`IndependentNoisyLinear` (*class in utils.ctorch.nn*), 15
- ## L
- `length` (*utils.ctorch.rl.ReplayBuffer property*), 34
`log_norm_pdf()` (*in module utils.ctorch.functional*), 4
`log_pi` (*utils.ctorch.rl.Trajectory attribute*), 35
`log_pi()` (*utils.ctorch.rl.BasePolicyNetwork method*), 26
`logit_product()` (*in module utils.ctorch.functional*), 4
`loss()` (*utils.ctorch.rl.BaseRLModel method*), 29
- ## M
- `masked_select()` (*in module utils.ctorch.padding*), 21
`mean()` (*utils.ctorch.nn.GaussianLinear method*), 14
`MetricFormatter` (*class in utils.ctorch.metrics*), 8
`mmd_distance()` (*in module utils.ctorch.functional*), 4
`module`
 utils, 1
 utils.clogging, 39
 utils.cprint, 41
 utils.ctorch, 3
 utils.ctorch.device, 3
 utils.ctorch.functional, 4
 utils.ctorch.metrics, 7
 utils.ctorch.nn, 9
 utils.ctorch.ops, 19
 utils.ctorch.padding, 20
 utils.ctorch.rl, 25
 utils.parser, 43
 utils.sampler, 37
`Module` (*class in utils.ctorch.nn*), 15
`MonotonicLinear` (*class in utils.ctorch.nn*), 16
`MultivariateNormalClass` (*class in utils.ctorch.nn*), 16
- ## N
- `name` (*utils.ctorch.device.GpuStat attribute*), 3
`ndcg_score()` (*in module utils.ctorch.metrics*), 9
`next_state` (*utils.ctorch.rl.Trajectory attribute*), 35
`norm_pdf()` (*in module utils.ctorch.functional*), 5
`normalize_trajectory()`
 (*utils.ctorch.rl.BasePolicyNetwork method*), 26

`num_parameters` (*utils.ctorch.nn.Module* property), 15

P

`pack_padded_sequence_right()` (in module *utils.ctorch.padding*), 22

`packed_binary_op()` (in module *utils.ctorch.padding*), 22

`packed_concat()` (in module *utils.ctorch.padding*), 22

`packed_forward()` (in module *utils.ctorch.padding*), 23

`packed_unary_op()` (in module *utils.ctorch.padding*), 23

`pad_packed_sequence_right()` (in module *utils.ctorch.padding*), 23

`parse_all_args()` (in module *utils.parser*), 44

`pd()` (*utils.ctorch.nn.CholeskyTrilLinear* method), 10

`pi()` (*utils.ctorch.rl.BasePolicyNetwork* method), 26

`policy_parameters()` (*utils.ctorch.rl.BasePolicyNetwork* method), 26

`prepend_left()` (in module *utils.ctorch.padding*), 23

`PrioritizedReplayBuffer` (class in *utils.ctorch.rl*), 32

Q

`Q()` (*utils.ctorch.rl.BaseValueNetwork* method), 30

`Q_all()` (*utils.ctorch.rl.BaseQNetwork* method), 27

R

`rbf_kernel()` (in module *utils.ctorch.functional*), 5

`ReplayBuffer` (class in *utils.ctorch.rl*), 33

`reset()` (*utils.ctorch.metrics.BatchedMetric* method), 7

`reward` (*utils.ctorch.rl.Trajectory* attribute), 35

`RotaryTemporalEmbedding` (class in *utils.ctorch.nn*), 17

`run_episode()` (in module *utils.ctorch.rl*), 35

S

`sample_index()` (*utils.ctorch.rl.PrioritizedReplayBuffer* method), 33

`sample_index()` (*utils.ctorch.rl.ReplayBuffer* method), 34

`set_weight()` (*utils.ctorch.rl.PrioritizedReplayBuffer* method), 33

`setup_target()` (*utils.ctorch.rl.BasePolicyNetwork* method), 26

`setup_target()` (*utils.ctorch.rl.TargetNetworkMixin* method), 34

`SinusoidalTemporalEmbedding` (class in *utils.ctorch.nn*), 17

`size` (*utils.ctorch.rl.CircularTensor* property), 32

`state` (*utils.ctorch.rl.Trajectory* attribute), 35

`store()` (*utils.ctorch.rl.PrioritizedReplayBuffer* method), 33

`store()` (*utils.ctorch.rl.ReplayBuffer* method), 34

T

`target` (*utils.ctorch.rl.TargetNetworkMixin* property), 34

`TargetNetworkMixin` (class in *utils.ctorch.rl*), 34

`td_step()` (*utils.ctorch.rl.BaseQNetwork* method), 29

`td_step()` (*utils.ctorch.rl.BaseValueNetwork* method), 31

`tensor_fields` (*utils.ctorch.rl.Trajectory* attribute), 35

`torch_step()` (in module *utils.ctorch.rl*), 35

`total_embedding_size` (*utils.ctorch.nn.FeatureEmbedding* property), 13

`total_gb` (*utils.ctorch.device.GpuStat* attribute), 3

`total_reward` (*utils.ctorch.rl.Trajectory* attribute), 35

`Trajectory` (class in *utils.ctorch.rl*), 34

`TransformerDecoderLayer` (class in *utils.ctorch.nn*), 18

`TransformerEncoderLayer` (class in *utils.ctorch.nn*), 19

`truncate_left()` (in module *utils.ctorch.padding*), 24

`truncate_right()` (in module *utils.ctorch.padding*), 24

U

`unpad_sequence_right()` (in module *utils.ctorch.padding*), 24

`update()` (*utils.ctorch.metrics.MetricFormatter* method), 8

`update_target()` (*utils.ctorch.rl.TargetNetworkMixin* method), 34

`utils`

module, 1

`utils.clogging`

module, 39

`utils.cprint`

module, 41

`utils.ctorch`

module, 3

`utils.ctorch.device`

module, 3

`utils.ctorch.functional`

module, 4

`utils.ctorch.metrics`

module, 7

`utils.ctorch.nn`

module, 9

`utils.ctorch.ops`

module, 19

`utils.ctorch.padding`

module, 20

`utils.ctorch.rl`

module, 25

`utils.parser`

module, 43

`utils.sampler`

module, 37

V

`V()` (*utils.ctorch.rl.BaseQNetwork method*), [27](#)

`V()` (*utils.ctorch.rl.BaseValueNetwork method*), [30](#)

`value_model` (*utils.ctorch.rl.BasePolicyNetwork property*), [26](#)

`value_parameters()` (*utils.ctorch.rl.BasePolicyNetwork method*), [26](#)

W

`wasserstein_distance()` (*in module utils.ctorch.functional*), [6](#)