SecML-Torch

Release 1.0.0

Maura Pintor, Luca Demetrio

May 13, 2024

INTRO:

CHAPTER

ONE

SECML-TORCH: A LIBRARY FOR ROBUSTNESS EVALUATION OF DEEP LEARNING MODELS

SecML-Torch (SecMLT) is an open-source Python library designed to facilitate research in the area of Adversarial Machine Learning (AML) and robustness evaluation. The library provides a simple yet powerful interface for generating various types of adversarial examples, as well as tools for evaluating the robustness of machine learning models against such attacks.

1.1 Installation

You can install SecMLT via pip:

```
pip install secml-torch
```

This will install the core version of SecMLT, including only the main functionalities such as native implementation of attacks and PyTorch wrappers.

1.1.1 Install with extras

The library can be installed together with other plugins that enable further functionalities.

- Foolbox, a Python toolbox to create adversarial examples.
- Tensorboard, a visualization toolkit for machine learning experimentation.

Install one or more extras with the command:

```
pip install secml-torch[foolbox,tensorboard]
```

1.2 Key Features

- **Built for Deep Learning:** SecMLT is compatible with the popular machine learning framework PyTorch.
- **Various types of adversarial attacks:** SecMLT includes support for a wide range of attack methods (evasion, poisoning, ...) such as different implementations imported from popular AML libraries (Foolbox, Adversarial Library).
- **Customizable attacks:** SecMLT offers several levels of analysis for the models, including modular implementations of existing attacks to extend with different loss functions, optimizers, and more.

- **Attack debugging:** Built-in debugging of evaluations by logging events and metrics along the attack runs (even on Tensorboard).

1.3 Usage

Here's a brief example of using SecMLT to evaluate the robustness of a trained classifier:

```
from secmlt.adv.evasion.pgd import PGD
from secmlt.metrics.classification import Accuracy
from secmlt.models.pytorch.base_pytorch_nn import BasePytorchClassifier


model = ...
torch_data_loader = ...

# Wrap model
model = BasePytorchClassifier(model)

# create and run attack
attack = PGD(
    perturbation_model="l2",
    epsilon=0.4,
    num_steps=100,
    step_size=0.01,
)

adversarial_loader = attack(model, torch_data_loader)

# Test accuracy on adversarial examples
robust_accuracy = Accuracy()(model, adversarial_loader)
```

For more detailed usage instructions and examples, please refer to the official documentation or to the examples.

1.4 Contributing

We welcome contributions from the research community to expand the library's capabilities or add new features. If you would like to contribute to SecMLT, please follow our contribution guidelines.

1.4.1 Contributors

1.5 Acknowledgements

SecML has been partially developed with the support of European Union's ELSA – European Lighthouse on Secure and Safe AI, Horizon Europe, grant agreement No. 101070617, and Sec4AI4Sec - Cybersecurity for AI-Augmented Systems, Horizon Europe, grant agreement No. 101120393.

CHAPTER

TWO

SECML-TORCH: A LIBRARY FOR ROBUSTNESS EVALUATION OF DEEP LEARNING MODELS

SecML-Torch (SecMLT) is an open-source Python library designed to facilitate research in the area of Adversarial Machine Learning (AML) and robustness evaluation. The library provides a simple yet powerful interface for generating various types of adversarial examples, as well as tools for evaluating the robustness of machine learning models against such attacks.

2.1 Installation

You can install SecMLT via pip:

```
pip install secml-torch
```

This will install the core version of SecMLT, including only the main functionalities such as native implementation of attacks and PyTorch wrappers.

2.1.1 Install with extras

The library can be installed together with other plugins that enable further functionalities.

- Foolbox, a Python toolbox to create adversarial examples.
- Tensorboard, a visualization toolkit for machine learning experimentation.

Install one or more extras with the command:

```
pip install secml-torch[foolbox,tensorboard]
```

2.2 Key Features

- **Built for Deep Learning:** SecMLT is compatible with the popular machine learning framework PyTorch.
- **Various types of adversarial attacks:** SecMLT includes support for a wide range of attack methods (evasion, poisoning, ...) such as different implementations imported from popular AML libraries (Foolbox, Adversarial Library).
- **Customizable attacks:** SecMLT offers several levels of analysis for the models, including modular implementations of existing attacks to extend with different loss functions, optimizers, and more.

- **Attack debugging:** Built-in debugging of evaluations by logging events and metrics along the attack runs (even on Tensorboard).

2.3 Usage

Here's a brief example of using SecMLT to evaluate the robustness of a trained classifier:

```
from secmlt.adv.evasion.pgd import PGD
from secmlt.metrics.classification import Accuracy
from secmlt.models.pytorch.base_pytorch_nn import BasePytorchClassifier


model = ...
torch_data_loader = ...

# Wrap model
model = BasePytorchClassifier(model)

# create and run attack
attack = PGD(
    perturbation_model="l2",
    epsilon=0.4,
    num_steps=100,
    step_size=0.01,
)

adversarial_loader = attack(model, torch_data_loader)

# Test accuracy on adversarial examples
robust_accuracy = Accuracy()(model, adversarial_loader)
```

For more detailed usage instructions and examples, please refer to the official documentation or to the examples.

2.4 Contributing

We welcome contributions from the research community to expand the library's capabilities or add new features. If you would like to contribute to SecMLT, please follow our contribution guidelines.

2.4.1 Contributors

2.5 Acknowledgements

SecML has been partially developed with the support of European Union's ELSA – European Lighthouse on Secure and Safe AI, Horizon Europe, grant agreement No. 101070617, and Sec4AI4Sec - Cybersecurity for AI-Augmented Systems, Horizon Europe, grant agreement No. 101120393.

CHAPTER

THREE

SECMLT.ADV PACKAGE

3.1 Subpackages

3.1.1 secmlt.adv.evasion package

Subpackages

secmlt.adv.evasion.aggregators package

Submodules

secmlt.adv.evasion.aggregators.ensemble module

Ensemble metrics for getting best results across multiple attacks.

class secmlt.adv.evasion.aggregators.ensemble.**Ensemble**

Bases: ABC

Abstract class for creating an ensemble metric.

__call__(*model:* BaseModel, *data_loader: torch.utils.data.DataLoader*, *adv_loaders: list[torch.utils.data.DataLoader]*) → torch.utils.data.DataLoader.torch.Tuple.torch.Tensor

Get the worst-case of the metric with the given implemented criterion.

Parameters

- **model** (BaseModel) – Model to use for predictions.
- **data_loader** (*DataLoader*) – Test dataloader.
- **adv_loaders** (*list[DataLoader]*) – List of dataloaders returned by multiple attacks.

Returns
The worst-case metric computed on the multiple attacks.

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

abstract _get_best(*model:* BaseModel, *samples: torch.Tensor*, *labels: torch.Tensor*, *x_adv: torch.Tensor*, *best_x_adv: torch.Tensor*) → torch.Tensor

Get the best result from multiple attacks.

Parameters

- **model** (BaseModel) – Model to use to predict.

- **samples** (*torch.Tensor*) – Input samples.
- **labels** (*torch.Tensor*) – Labels for the samples.
- **x_adv** (*torch.Tensor*) – Adversarial examples.
- **best_x_adv** (*torch.Tensor*) – Best adversarial examples found so far.

Returns
Best adversarial examples between the current x_adv and the ones already tested on the given model.

Return type
torch.Tensor

class secmlt.adv.evasion.aggregators.ensemble.**FixedEpsilonEnsemble**(*loss_fn: torch.nn.Module*, *maximize: bool = True*, *y_target: torch.Tensor | None = None*)

Bases: *Ensemble*

Wrapper for ensembling results of multiple fixed-epsilon attacks.

__init__(*loss_fn: torch.nn.Module*, *maximize: bool = True*, *y_target: torch.Tensor | None = None*) → None

Create fixed epsilon ensemble.

Parameters

- **loss_fn** (*torch.nn.Module*) – Loss function to maximize (or minimize).
- **maximize** (*bool, optional*) – If True maximizes the loss otherwise it minimizes it, by default True.
- **y_target** (*torch.Tensor | None, optional*) – Target label for targeted attacks, None for untargeted, by default None.

_get_best(*model: BaseModel*, *samples: torch.Tensor*, *labels: torch.Tensor*, *x_adv: torch.Tensor*, *best_x_adv: torch.Tensor*) → torch.Tensor

Get the adversarial examples with maximum (or minimum) loss.

Parameters

- **model** (*BaseModel*) – Model to use to predict.
- **samples** (*torch.Tensor*) – Input samples.
- **labels** (*torch.Tensor*) – Labels for the samples.
- **x_adv** (*torch.Tensor*) – Adversarial examples.
- **best_x_adv** (*torch.Tensor*) – Best adversarial examples found so far.

Returns
The maximum-loss adversarial examples found so far.

Return type
torch.Tensor

class secmlt.adv.evasion.aggregators.ensemble.**MinDistanceEnsemble**(*perturbation_model: str*)

Bases: *Ensemble*

Wrapper for ensembling results of multiple minimum-distance attacks.

__init__(*perturbation_model: str*) → None

Create MinDistance Ensemble.

Parameters
perturbation_model (*str*) – Perturbation model to use to compute the distance.

_get_best(*model:* BaseModel, *samples: torch.Tensor*, *labels: torch.Tensor*, *x_adv: torch.Tensor*, *best_x_adv: torch.Tensor*) → torch.Tensor

Get the adversarial examples with minimal perturbation.

Parameters

- **model** (BaseModel) – Model to use to predict.
- **samples** (*torch.Tensor*) – Input samples.
- **labels** (*torch.Tensor*) – Labels for the samples.
- **x_adv** (*torch.Tensor*) – Adversarial examples.
- **best_x_adv** (*torch.Tensor*) – Best adversarial examples found so far.

Returns
The minimum-distance adversarial examples found so far.

Return type
torch.Tensor

Module contents

Aggregator functions for multiple attacks or multiple attack runs.

secmlt.adv.evasion.foolbox_attacks package

Submodules

secmlt.adv.evasion.foolbox_attacks.foolbox_base module

Generic wrapper for Foolbox evasion attacks.

class secmlt.adv.evasion.foolbox_attacks.foolbox_base.**BaseFoolboxEvasionAttack**(*foolbox_attack: type[foolbox.attacks.base.Attack*, *epsilon: float = torch.inf*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: type[secmlt.trackers.tracker.Tra | None = None*)

Bases: *BaseEvasionAttack*

Generic wrapper for Foolbox Evasion attacks.

__init__(*foolbox_attack: type[foolbox.attacks.base.Attack]*, *epsilon: float = torch.inf*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: type[secmlt.trackers.tracker.Tracker] | None = None*) → None

Wrap Foolbox attacks.

Parameters

- **foolbox_attack** (*Type[Attack]*) – Foolbox attack class to wrap.
- **epsilon** (*float, optional*) – Perturbation constraint, by default torch.inf.
- **y_target** (*int | None, optional*) – Target label for the attack, None if untargeted, by default None.
- **lb** (*float, optional*) – Lower bound of the input space, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the input space, by default 1.0.
- **trackers** (*type[TRACKER_TYPE] | None, optional*) – Trackers for the attack (unallowed in Foolbox), by default None.

secmlt.adv.evasion.foolbox_attacks.foolbox_pgd module

Wrapper of the PGD attack implemented in Foolbox.

class secmlt.adv.evasion.foolbox_attacks.foolbox_pgd.**PGDFoolbox**(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, ***kwargs*)

Bases: *BaseFoolboxEvasionAttack*

Wrapper of the Foolbox implementation of the PGD attack.

__init__(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, ***kwargs*) → None

Create PGD attack with Foolbox backend.

Parameters

- **perturbation_model** (*str*) – Perturbation model for the attack.
- **epsilon** (*float*) – Maximum perturbation allowed.
- **num_steps** (*int*) – Number of iterations for the attack.
- **step_size** (*float*) – Attack step size.
- **random_start** (*bool*) – True for randomly initializing the perturbation.
- **y_target** (*int | None, optional*) – Target label for the attack, None for untargeted, by default None.
- **lb** (*float, optional*) – Lower bound of the input space, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the input space, by default 1.0.

static **get_perturbation_models**() → set[str]

Check the perturbation models implemented for this attack.

Returns
The list of perturbation models implemented for this attack.

Return type
set[str]

Module contents

Wrappers of Foolbox library for evasion attacks.

Submodules

secmlt.adv.evasion.base_evasion_attack module

Base classes for implementing attacks and wrapping backends.

class secmlt.adv.evasion.base_evasion_attack.**BaseEvasionAttack**

Bases: object

Base class for evasion attacks.

__call__(*model:* BaseModel, *data_loader: torch.utils.data.DataLoader*) → torch.utils.data.DataLoader

Compute the attack against the model, using the input data.

Parameters

- **model** (BaseModel) – Model to test.
- **data_loader** (*DataLoader*) – Test dataloader.

Returns
Dataloader with adversarial examples and original labels.

Return type
DataLoader

classmethod check_perturbation_model_available(*perturbation_model: str*) → bool

Check whether the given perturbation model is available for the attack.

Parameters
perturbation_model (*str*) – A perturbation model.

Returns
True if the attack implements the given perturbation model.

Return type
bool

Raises
NotImplementedError – Raises NotImplementedError if not implemented in the inherited class.

abstract static get_perturbation_models() → set[str]

Check the perturbation models implemented for the given attack.

Returns
The set of perturbation models for which the attack is implemented.

Return type
set[str]

Raises
NotImplementedError – Raises NotImplementedError if not implemented in the inherited class.

property trackers: list[secmlt.trackers.tracker.Tracker] | None

Get the trackers set for this attack.

Returns
Trackers set for the attack, if any.

Return type
list[TRACKER_TYPE] | None

class secmlt.adv.evasion.base_evasion_attack.**BaseEvasionAttackCreator**

Bases: object

Generic creator for attacks.

classmethod check_backend_available(*backend: str*) → bool

Check if a given backend is available for the attack.

Parameters
backend (*str*) – Backend string.

Returns
True if the given backend is implemented.

Return type
bool

Raises
NotImplementedError – Raises NotImplementedError if the requested backend is not in the list of the possible backends (check secmlt.adv.backends).

abstract static get_backends() → set[str]

Get the available backends for the given attack.

Returns
Set of implemented backends available for the attack.

Return type
set[str]

Raises
NotImplementedError – Raises NotImplementedError if not implemented in the inherited class.

classmethod get_foolbox_implementation() → *BaseEvasionAttack*

Get the Foolbox implementation of the attack.

Returns
Foolbox implementation of the attack.

Return type
BaseEvasionAttack

Raises
ImportError – Raises ImportError if Foolbox extra is not installed.

classmethod get_implementation(*backend: str*) → *BaseEvasionAttack*

Get the implementation of the attack with the given backend.

Parameters
backend (*str*) – The backend for the attack. See secmlt.adv.backends for available backends.

Returns
Attack implementation.

Return type
BaseEvasionAttack

secmlt.adv.evasion.modular_attack module

Implementation of modular iterative attacks with customizable components.

class secmlt.adv.evasion.modular_attack.**ModularEvasionAttackFixedEps**(*y_target: int | None*, *num_steps: int*, *step_size: float*, *loss_function: str | torch.nn.Module*, *optimizer_cls: str | partial[torch.optim.Optimizer]*, *manipulation_function:* Manipulation, *initializer:* Initializer, *gradient_processing:* GradientProcessing, *trackers: list[*Tracker*] |* Tracker *| None = None*)

Bases: *BaseEvasionAttack*

Modular evasion attack for fixed-epsilon attacks.

__init__(*y_target: int | None*, *num_steps: int*, *step_size: float*, *loss_function: str | torch.nn.Module*, *optimizer_cls: str | partial[torch.optim.Optimizer]*, *manipulation_function:* Manipulation, *initializer:* Initializer, *gradient_processing:* GradientProcessing, *trackers: list[*Tracker*] |* Tracker *| None = None*) → None

Create modular evasion attack.

Parameters

- **y_target** (*int | None*) – Target label for the attack, None for untargeted.
- **num_steps** (*int*) – Number of iterations for the attack.
- **step_size** (*float*) – Attack step size.
- **loss_function** (*str | torch.nn.Module*) – Loss function to minimize.
- **optimizer_cls** (*str | partial[Optimizer]*) – Algorithm for solving the attack optimization problem.
- **manipulation_function** (Manipulation) – Manipulation function to perturb the inputs.
- **initializer** (Initializer) – Initialization for the perturbation delta.
- **gradient_processing** (GradientProcessing) – Gradient transformation function.
- **trackers** (*list[Tracker] | Tracker | None, optional*) – Trackers for logging, by default None.

Raises
ValueError – Raises ValueError if the loss is not in allowed list of loss functions.

classmethod get_perturbation_models() → set[str]

Check if a given perturbation model is implemented.

Returns
Set of perturbation models available for this attack.

Return type
set[str]

secmlt.adv.evasion.perturbation_models module

Implementation of perturbation models for perturbations of adversarial examples.

class secmlt.adv.evasion.perturbation_models.**LpPerturbationModels**

Bases: object

Lp perturbation models.

L0 = 'l0'

L1 = 'l1'

L2 = 'l2'

LINF = 'linf'

classmethod get_p(*perturbation_model: str*) → float

Get the float representation of p from the given string.

Parameters
perturbation_model (*str*) – One of the strings defined in PerturbationModels.pert_models.

Returns
The float representation of p, to use. e.g., in torch.norm(p=...).

Return type
float

Raises
ValueError – Raises ValueError if the norm given is not in PerturbationModels.pert_models

classmethod is_perturbation_model_available(*perturbation_model: str*) → bool

Check availability of the perturbation model requested.

Parameters
perturbation_model (*str*) – A perturbation model as a string.

Returns
True if the perturbation model is found in PerturbationModels.pert_models.

Return type
bool

pert_models: ClassVar[dict[str, float]] = {'l0': 0, 'l1': 1, 'l2': 2, 'linf': inf}

secmlt.adv.evasion.pgd module

Implementations of the Projected Gradient Descent evasion attack.

class secmlt.adv.evasion.pgd.**PGD**(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool = False*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *backend: str = 'foolbox'*, *trackers: list[Tracker] | None = None*, ***kwargs*)

Bases: *BaseEvasionAttackCreator*

Creator for the Projected Gradient Descent (PGD) attack.

static **__new__**(*cls*, *perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool = False*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *backend: str = 'foolbox'*, *trackers: list[Tracker] | None = None*, ***kwargs*) → *BaseEvasionAttack*

Create the PGD attack.

Parameters

- **perturbation_model** (*str*) – Perturbation model for the attack. Available: 1, 2, inf.
- **epsilon** (*float*) – Radius of the constraint for the Lp ball.
- **num_steps** (*int*) – Number of iterations for the attack.
- **step_size** (*float*) – Attack step size.
- **random_start** (*bool, optional*) – Whether to use a random initialization onto the Lp ball, by default False.
- **y_target** (*int | None, optional*) – Target label for a targeted attack, None for untargeted attack, by default None.
- **lb** (*float, optional*) – Lower bound of the input space, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the input space, by default 1.0.
- **backend** (*str, optional*) – Backend to use to run the attack, by default Backends.FOOLBOX
- **trackers** (*list[Tracker] | None, optional*) – Trackers to check various attack metrics (see secmlt.trackers), available only for native implementation, by default None.

Returns

PGD attack instance.

Return type

BaseEvasionAttack

static **get_backends**() → list[str]

Get available implementations for the PGD attack.

class secmlt.adv.evasion.pgd.**PGDFoolbox**(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: list[Tracker] | None = None*, ***kwargs*)

Bases: *BaseFoolboxEvasionAttack*

Foolbox implementation of the PGD attack.

__init__(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: list[Tracker] | None = None*, ***kwargs*) → None

Create Foolbox PGD attack.

Parameters

- **perturbation_model** (*str*) – Perturbation model for the attack. Available: 1, 2, inf.
- **epsilon** (*float*) – Radius of the constraint for the Lp ball.
- **num_steps** (*int*) – Number of iterations for the attack.
- **step_size** (*float*) – Attack step size.
- **random_start** (*bool*) – Whether to use a random initialization onto the Lp ball.
- **y_target** (*int | None, optional*) – Target label for a targeted attack, None for untargeted attack, by default None.
- **lb** (*float, optional*) – Lower bound of the input space, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the input space, by default 1.0.
- **trackers** (*list[Tracker] | None, optional*) – Trackers to check various attack metrics (see secmlt.trackers), available only for native implementation, by default None.

Raises

NotImplementedError – Raises NotImplementedError if the requested perturbation model is not defined for this attack.

static get_perturbation_models() → set[str]

Check the perturbation models implemented for the given attack.

Returns

The set of perturbation models for which the attack is implemented.

Return type

set[str]

Raises

NotImplementedError – Raises NotImplementedError if not implemented in the inherited class.

class secmlt.adv.evasion.pgd.**PGDNative**(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: list[Tracker] | None = None*, ***kwargs*)

Bases: *ModularEvasionAttackFixedEps*

Native implementation of the Projected Gradient Descent attack.

__init__(*perturbation_model: str*, *epsilon: float*, *num_steps: int*, *step_size: float*, *random_start: bool*, *y_target: int | None = None*, *lb: float = 0.0*, *ub: float = 1.0*, *trackers: list[Tracker] | None = None*, ***kwargs*) → None

Create Native PGD attack.

Parameters

- **perturbation_model** (*str*) – Perturbation model for the attack. Available: 1, 2, inf.
- **epsilon** (*float*) – Radius of the constraint for the Lp ball.
- **num_steps** (*int*) – Number of iterations for the attack.

- **step_size** (*float*) – Attack step size.
- **random_start** (*bool*) – Whether to use a random initialization onto the Lp ball.
- **y_target** (*int | None, optional*) – Target label for a targeted attack, None for untargeted attack, by default None.
- **lb** (*float, optional*) – Lower bound of the input space, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the input space, by default 1.0.
- **trackers** (*list[Tracker] | None, optional*) – Trackers to check various attack metrics (see secmlt.trackers), available only for native implementation, by default None.

Module contents

Evasion attack functionalities.

3.2 Submodules

3.3 secmlt.adv.backends module

Available backends for running adversarial attacks.

class secmlt.adv.backends.**Backends**

Bases: object

Available backends.

FOOLBOX = 'foolbox'

NATIVE = 'native'

3.4 Module contents

Adversarial functionalities.

CHAPTER

FOUR

SECMLT.DATA PACKAGE

4.1 Submodules

4.2 secmlt.data.distributions module

Implementation for uncommon distributions.

class secmlt.data.distributions.**Distribution**

Bases: ABC

Abstract class for distributions.

abstract sample(*shape: torch.Size*) → torch.Tensor

Sample from the distribution.

This method generates a sample from the distribution, with the specified shape. If no shape is specified, a single sample is returned.

Parameters

shape (*torch.Size, optional*) – The shape of the sample to be generated. Default is torch.Size(), which corresponds to a single sample.

Returns

A tensor of samples from the distribution, with the specified shape.

Return type

torch.Tensor

class secmlt.data.distributions.**GeneralizedNormal**

Bases: *Distribution*

Generalized normal distribution.

$$f(x;\mu,\alpha,\beta) = \frac{\beta}{2\alpha\Gamma(1/\beta)} e^{-(|x-\mu|/\alpha)^{\beta}}$$

where *mu* is the location parameter, *alpha* is the scale parameter, and *beta* is the shape parameter.

sample(*shape: torch.Size*, *p: float = 2*) → torch.Tensor

Sample from the generalized normal distribution.

This method generates a sample from the generalized normal distribution, with shape parameter p. The shape of the output is determined by the *shape* parameter.

Parameters

- **shape** (*torch.Size*) – The shape of the sample to be generated.

- **p** (*float, optional*) – The shape parameter of the generalized normal distribution. Default is 2.

Returns
A tensor of samples from the generalized normal distribution.

Return type
torch.Tensor

Examples

```
>>> dist = GeneralizedNormal()
>>> sample = dist.sample((3, 4))
```

class secmlt.data.distributions.**Rademacher**

Bases: *Distribution*

Samples from Rademacher distribution (-1, 1) with equal probability.

sample(*shape: torch.Size*) → torch.Tensor

Sample from the Rademacher distribution.

This method generates a sample from the Rademacher distribution, where each sample is either -1 or 1 with equal probability. The shape of the output is determined by the *shape* parameter.

Parameters
shape (*torch.Size*) – The shape of the sample to be generated.

Returns
A tensor of samples from the Rademacher distribution, with values -1 or 1.

Return type
torch.Tensor

Examples

```
>>> dist = Rademacher()
>>> sample = dist.sample((3, 4))
```

4.3 secmlt.data.lp_uniform_sampling module

Implementation of Lp uniform sampling.

class secmlt.data.lp_uniform_sampling.**LpUniformSampling**(*p: str = 'l2'*)

Bases: object

Uniform sampling from the unit Lp ball.

This class provides a method for sampling uniformly from the unit Lp ball, where Lp is a norm defined by a parameter *p*. The class supports sampling from the L0, L2, and Linf norms.

The sampling method is based on the following reference: https://arxiv.org/abs/math/0503650

Variables
p (*str*) – The norm to use for sampling. Must be one of 'l0', 'l1', 'l2', 'linf'.

__init__(*p: str = 'l2'*) → None

Initialize the LpUniformSampling object.

Parameters
p (*str, optional*) – The norm to use for sampling. Must be one of 'L0', 'L2', or 'Linf'. Default is 'L2'.

sample(*num_samples: int = 1*, *dim: int = 2*) → torch.Tensor

Sample uniformly from the unit Lp ball.

This method generates a specified number of samples from the unit Lp ball, where Lp is a norm defined by the *p* parameter. The samples are generated using the algorithm described in the class documentation.

Parameters

- **num_samples** (*int*) – The number of samples to generate.
- **dim** (*int*) – The dimension of the samples.

Returns
A tensor of samples from the unit Lp ball, with shape *(num_samples, dim)*.

Return type
torch.Tensor

sample_like(*x: torch.Tensor*) → torch.Tensor

Sample from the unit Lp ball with the same shape as a given tensor.

Parameters
x (*torch.Tensor*) – The input tensor whose shape is used to determine the shape of the samples.

Returns
A tensor of samples from the unit Lp ball, with the same shape as the input tensor *x*.

Return type
torch.Tensor

4.4 Module contents

Functionalities for handling data.

CHAPTER

FIVE

SECMLT.MANIPULATIONS PACKAGE

5.1 Submodules

5.2 secmlt.manipulations.manipulation module

Manipulations for perturbing input samples.

class secmlt.manipulations.manipulation.**AdditiveManipulation**(*domain_constraints: list[Constraint]*, *perturbation_constraints: list[Constraint]*)

Bases: *Manipulation*

Additive manipulation for input data.

class secmlt.manipulations.manipulation.**Manipulation**(*domain_constraints: list[Constraint]*, *perturbation_constraints: list[Constraint]*)

Bases: ABC

Abstract class for manipulations.

__call__(*x: torch.Tensor*, *delta: torch.Tensor*) → tuple[torch.Tensor, torch.Tensor]

Apply the manipulation to the input data.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **delta** (*torch.Tensor*) – Perturbation to apply.

Returns

Perturbed data and perturbation after the application of constraints.

Return type

tuple[torch.Tensor, torch.Tensor]

__init__(*domain_constraints: list[Constraint]*, *perturbation_constraints: list[Constraint]*) → None

Create manipulation object.

Parameters

- **domain_constraints** (*list[Constraint]*) – Constraints for the domain bounds (x_adv).
- **perturbation_constraints** (*list[Constraint]*) – Constraints for the perturbation (delta).

abstract _apply_manipulation(*x: torch.Tensor*, *delta: torch.Tensor*) → torch.Tensor

Apply the manipulation.

Parameters

- **x** (*torch.Tensor*) – Input samples.
- **delta** (*torch.Tensor*) – Manipulation to apply.

Returns
Perturbed samples.

Return type
torch.Tensor

5.3 Module contents

Functionalities for applying manipulations to input data.

CHAPTER

SIX

SECMLT.METRICS PACKAGE

6.1 Submodules

6.2 secmlt.metrics.classification module

Classification metrics for machine-learning models and for attack performance.

class secmlt.metrics.classification.**Accuracy**

Bases: object

Class for computing accuracy of a model on a dataset.

__call__(*model:* BaseModel, *dataloader: torch.utils.data.DataLoader*) → torch.Tensor

Compute the metric on a single attack run or a dataloader.

Parameters

- **model** (BaseModel) – Model to use for prediction.
- **dataloader** (*DataLoader*) – A dataloader, can be the result of an attack or a generic test dataloader.

Returns
The metric computed on the given dataloader.

Return type
torch.Tensor

__init__() → None

Create Accuracy metric.

class secmlt.metrics.classification.**AccuracyEnsemble**

Bases: *Accuracy*

Robust accuracy of a model on multiple attack runs.

__call__(*model:* BaseModel, *dataloaders: list[torch.utils.data.DataLoader]*) → torch.Tensor

Compute the metric on an ensemble of attacks from their results.

Parameters

- **model** (BaseModel) – Model to use for prediction.
- **dataloaders** (*list[DataLoader]*) – List of loaders returned from multiple attack runs.

Returns
The metric computed across multiple attack runs.

Return type
torch.Tensor

class secmlt.metrics.classification.**AttackSuccessRate**(*y_target: float | torch.Tensor | None = None*)

Bases: *Accuracy*

Single attack success rate from attack results.

__init__(*y_target: float | torch.Tensor | None = None*) → None

Create attack success rate metric.

Parameters
y_target (*float | torch.Tensor | None, optional*) – Target label for the attack, None for untargeted, by default None

class secmlt.metrics.classification.**EnsembleSuccessRate**(*y_target: float | torch.Tensor | None = None*)

Bases: *AccuracyEnsemble*

Worst-case success rate of multiple attack runs.

__init__(*y_target: float | torch.Tensor | None = None*) → None

Create ensemble success rate metric.

Parameters
y_target (*float | torch.Tensor | None, optional*) – Target label for the attack, None for untargeted,, by default None

secmlt.metrics.classification.**accuracy**(*y_pred: torch.Tensor*, *y_true: torch.Tensor*) → torch.Tensor

Compute the accuracy on a batch of predictions and targets.

Parameters

- **y_pred** (*torch.Tensor*) – Predictions from the model.
- **y_true** (*torch.Tensor*) – Target labels.

Returns
The percentage of predictions that match the targets.

Return type
torch.Tensor

6.3 Module contents

Metrics to evaluate machine learning models and attacks.

CHAPTER

SEVEN

SECMLT.MODELS PACKAGE

7.1 Subpackages

7.1.1 secmlt.models.data_processing package

Submodules

secmlt.models.data_processing.data_processing module

Interface for the data processing functionalities.

class secmlt.models.data_processing.data_processing.**DataProcessing**

Bases: ABC

Abstract data processing class.

__call__(*x: torch.Tensor*) → torch.Tensor

Apply the forward transformation.

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

Returns
The samples after transformation.

Return type
torch.Tensor

abstract invert(*x: torch.Tensor*) → torch.Tensor

Apply the inverted transform (if defined).

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

Returns
The samples in the input space before the transformation.

Return type
torch.Tensor

secmlt.models.data_processing.identity_data_processing module

Identity data processing, returns the samples as they are.

class secmlt.models.data_processing.identity_data_processing.**IdentityDataProcessing**

Bases: *DataProcessing*

Identity transformation.

_process(*x: torch.Tensor*) → torch.Tensor

Identity transformation. Returns the samples unchanged.

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

Returns
Unchanged samples.

Return type
torch.Tensor

invert(*x: torch.Tensor*) → torch.Tensor

Return the sample as it is.

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

Returns
Unchanged samples for identity inverse transformation.

Return type
torch.Tensor

Module contents

Functionalities for data transformations.

7.1.2 secmlt.models.pytorch package

Submodules

secmlt.models.pytorch.base_pytorch_nn module

Wrappers for PyTorch models.

class secmlt.models.pytorch.base_pytorch_nn.**BasePytorchClassifier**(*model: torch.nn.Module*, *preprocessing:* DataProcessing | *None* = *None*, *postprocessing:* DataProcessing | *None* = *None*, *trainer:* BasePyTorchTrainer | *None* = *None*)

Bases: *BaseModel*

Wrapper for PyTorch classifier.

__init__(*model: torch.nn.Module*, *preprocessing:* DataProcessing | *None* = *None*, *postprocessing:* DataProcessing | *None* = *None*, *trainer:* BasePyTorchTrainer | *None* = *None*) → None

Create wrapped PyTorch classifier.

Parameters

- **model** (*torch.nn.Module*) – PyTorch model.
- **preprocessing** (DataProcessing, *optional*) – Preprocessing to apply before the forward., by default None.
- **postprocessing** (DataProcessing, *optional*) – Postprocessing to apply after the forward, by default None.
- **trainer** (BasePyTorchTrainer, *optional*) – Trainer object to train the model, by default None.

_decision_function(*x: torch.Tensor*) → torch.Tensor

Compute decision function of the model.

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

Returns
Output scores from the model.

Return type
torch.Tensor

gradient(*x: torch.Tensor*, *y: int*) → torch.Tensor

Compute batch gradients of class y w.r.t. x.

Parameters

- **x** (*torch.Tensor*) – Input samples.
- **y** (*int*) – Class label.

Returns
Gradient of class y w.r.t. input x.

Return type
torch.Tensor

property model: torch.nn.Module

Get the wrapped instance of PyTorch model.

Returns
Wrapped PyTorch model.

Return type
torch.nn.Module

predict(*x: torch.Tensor*) → torch.Tensor

Return the predicted class for the given samples.

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

Returns
Predicted class for the samples.

Return type
torch.Tensor

train(*dataloader: torch.utils.data.DataLoader*) → torch.nn.Module

Train the model with given dataloader, if trainer is set.

Parameters
dataloader (*DataLoader*) – Training PyTorch dataloader to use for training.

Returns
Trained PyTorch model.

Return type
torch.nn.Module

Raises
ValueError – Raises ValueError if the trainer is not set.

secmlt.models.pytorch.base_pytorch_trainer module

PyTorch model trainers.

class secmlt.models.pytorch.base_pytorch_trainer.**BasePyTorchTrainer**(*optimizer: torch.optim.Optimizer*, *epochs: int = 5*, *loss: torch.nn.Module | None = None*, *scheduler: torch.optim.lr_scheduler._LRScheduler | None = None*)

Bases: *BaseTrainer*

Trainer for PyTorch models.

__init__(*optimizer: torch.optim.Optimizer*, *epochs: int = 5*, *loss: torch.nn.Module | None = None*, *scheduler: torch.optim.lr_scheduler._LRScheduler | None = None*) → None

Create PyTorch trainer.

Parameters

- **optimizer** (*torch.optim.Optimizer*) – Optimizer to use for training the model.
- **epochs** (*int, optional*) – Number of epochs, by default 5.
- **loss** (*torch.nn.Module, optional*) – Loss to minimize, by default None.
- **scheduler** (*_LRScheduler, optional*) – Scheduler for the optimizer, by default None.

train(*model: torch.nn.Module*, *dataloader: torch.utils.data.DataLoader*) → torch.nn.Module

Train model with given loader.

Parameters

- **model** (*torch.nn.Module*) – Pytorch model to be trained.
- **dataloader** (*DataLoader*) – Train data loader.

Returns
Trained model.

Return type
torch.nn.Module

Module contents

PyTorch model wrappers.

7.2 Submodules

7.3 secmlt.models.base_model module

Basic wrapper for generic model.

class secmlt.models.base_model.**BaseModel**(*preprocessing:* DataProcessing | *None* = *None*, *postprocessing:* DataProcessing | *None* = *None*)

Bases: ABC

Basic model wrapper.

__call__(*x: torch.Tensor*) → torch.Tensor

Forward function of the model.

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

Returns
Model ouptut scores.

Return type
torch.Tensor

__init__(*preprocessing:* DataProcessing | *None* = *None*, *postprocessing:* DataProcessing | *None* = *None*) → None

Create base model.

Parameters

- **preprocessing** (DataProcessing, *optional*) – Preprocessing to apply before the forward, by default None.
- **postprocessing** (DataProcessing, *optional*) – Postprocessing to apply after the forward, by default None.

abstract _decision_function(*x: torch.Tensor*) → torch.Tensor

Specific decision function of the model (data already preprocessed).

Parameters
x (*torch.Tensor*) – Preprocessed input samples.

Returns
Model output scores.

Return type
torch.Tensor

decision_function(*x: torch.Tensor*) → torch.Tensor

Return the decision function from the model.

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

Returns
Model output scores.

Return type
torch.Tensor

abstract gradient(*x: torch.Tensor*, *y: int*) → torch.Tensor

Compute gradients of the score y w.r.t. x.

Parameters

- **x** (*torch.Tensor*) – Input samples.
- **y** (*int*) – Target score.

Returns
Input gradients of the target score y.

Return type
torch.Tensor

abstract predict(*x: torch.Tensor*) → torch.Tensor

Return output predictions for given model.

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

Returns
Predictions from the model.

Return type
torch.Tensor

abstract train(*dataloader: torch.utils.data.DataLoader*) → *BaseModel*

Train the model with the given dataloader.

Parameters
dataloader (*DataLoader*) – Train data loader.

7.4 secmlt.models.base_trainer module

Model trainers.

class secmlt.models.base_trainer.**BaseTrainer**

Bases: object

Abstract class for model trainers.

abstract train(*model:* BaseModel, *dataloader: torch.utils.data.DataLoader*) → *BaseModel*

Train a model with the given dataloader.

Parameters

- **model** (BaseModel) – Model to train.
- **dataloader** (*DataLoader*) – Training dataloader.

Returns
The trained model.

Return type
BaseModel

7.5 Module contents

Machine learning models and wrappers.

CHAPTER

EIGHT

SECMLT.OPTIMIZATION PACKAGE

8.1 Submodules

8.2 secmlt.optimization.constraints module

Constraints for tensors and the corresponding batch-wise projections.

class secmlt.optimization.constraints.**ClipConstraint**(*lb: float = 0.0*, *ub: float = 1.0*)

Bases: *Constraint*

Box constraint, usually for the input space.

__call__(*x: torch.Tensor*, **args*, ***kwargs*) → torch.Tensor

Call the projection function.

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

Returns
Tensor projected onto the box constraint.

Return type
torch.Tensor

__init__(*lb: float = 0.0*, *ub: float = 1.0*) → None

Create box constraint.

Parameters

- **lb** (*float, optional*) – Lower bound of the domain, by default 0.0.
- **ub** (*float, optional*) – Upper bound of the domain, by default 1.0.

class secmlt.optimization.constraints.**Constraint**

Bases: ABC

Generic constraint.

abstract __call__(*x: torch.Tensor*, **args*, ***kwargs*) → torch.Tensor

Project onto the constraint.

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

Returns
Tensor projected onto the constraint.

Return type
torch.Tensor

class secmlt.optimization.constraints.**L0Constraint**(*radius: float = 0.0*, *center: float = 0.0*)

Bases: *LpConstraint*

L0 constraint.

__init__(*radius: float = 0.0*, *center: float = 0.0*) → None

Create L0 constraint.

Parameters

- **radius** (*float, optional*) – Radius of the constraint, by default 0.0.
- **center** (*float, optional*) – Center of the constraint, by default 0.0.

project(*x: torch.Tensor*) → torch.Tensor

Project the samples onto the L0 constraint.

Returns the sample with the top-k components preserved, and the rest set to zero.

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

Returns
Samples projected onto L0 constraint.

Return type
torch.Tensor

class secmlt.optimization.constraints.**L1Constraint**(*radius: float = 0.0*, *center: float = 0.0*)

Bases: *LpConstraint*

L1 constraint.

__init__(*radius: float = 0.0*, *center: float = 0.0*) → None

Create L1 constraint.

Parameters

- **radius** (*float, optional*) – Radius of the constraint, by default 0.0.
- **center** (*float, optional*) – Center of the constraint, by default 0.0.

project(*x: torch.Tensor*) → torch.Tensor

Compute Euclidean projection onto the L1 ball for a batch.

Source: https://gist.github.com/tonyduan/1329998205d88c566588e57e3e2c0c55

min ||x - u||_2 s.t. ||u||_1 <= eps

Inspired by the corresponding numpy version by Adrien Gaidon.

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

Returns
Projected tensor.

Return type
torch.Tensor

Notes

The complexity of this algorithm is in O(dlogd) as it involves sorting x.

References

[1] Efficient Projections onto the l1-Ball for Learning in High Dimensions
John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra. International Conference on Machine Learning (ICML 2008)

class secmlt.optimization.constraints.**L2Constraint**(*radius: float = 0.0*, *center: float = 0.0*)

Bases: *LpConstraint*

L2 constraint.

__init__(*radius: float = 0.0*, *center: float = 0.0*) → None

Create L2 constraint.

Parameters

- **radius** (*float, optional*) – Radius of the constraint, by default 0.0.
- **center** (*float, optional*) – Center of the constraint, by default 0.0.

project(*x: torch.Tensor*) → torch.Tensor

Project onto the L2 constraint.

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

Returns
Tensor projected onto the L2 constraint.

Return type
torch.Tensor

class secmlt.optimization.constraints.**LInfConstraint**(*radius: float = 0.0*, *center: float = 0.0*)

Bases: *LpConstraint*

Linf constraint.

__init__(*radius: float = 0.0*, *center: float = 0.0*) → None

Create Linf constraint.

Parameters

- **radius** (*float, optional*) – Radius of the constraint, by default 0.0.
- **center** (*float, optional*) – Center of the constraint, by default 0.0.

project(*x: torch.Tensor*) → torch.Tensor

Project onto the Linf constraint.

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

Returns
Tensor projected onto the Linf constraint.

Return type
torch.Tensor

class secmlt.optimization.constraints.**LpConstraint**(*radius: float = 0.0*, *center: float = 0.0*, *p: str = 'linf'*)

Bases: *Constraint*, ABC

Abstract class for Lp constraint.

__call__(*x: torch.Tensor*, **args*, ***kwargs*) → torch.Tensor

Project the samples onto the Lp constraint.

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

Returns
Tensor projected onto the Lp constraint.

Return type
torch.Tensor

__init__(*radius: float = 0.0*, *center: float = 0.0*, *p: str = 'linf'*) → None

Create Lp constraint.

Parameters

- **radius** (*float, optional*) – Radius of the constraint, by default 0.0.
- **center** (*float, optional*) – Center of the constraint, by default 0.0.
- **p** (*str, optional*) – Value of p for Lp norm, by default LpPerturbationModels.LINF.

abstract project(*x: torch.Tensor*) → torch.Tensor

Project onto the Lp constraint.

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

Returns
Tensor projected onto the Lp constraint.

Return type
torch.Tensor

8.3 secmlt.optimization.gradient_processing module

Processing functions for gradients.

class secmlt.optimization.gradient_processing.**GradientProcessing**

Bases: ABC

Gradient processing base class.

abstract __call__(*grad: torch.Tensor*) → torch.Tensor

Process the gradient with the given transformation.

Parameters
grad (*torch.Tensor*) – Input gradients.

Returns
The processed gradients.

Return type
torch.Tensor

class secmlt.optimization.gradient_processing.**LinearProjectionGradientProcessing**(*perturbation_model: str = 'l2'*)

Bases: *GradientProcessing*

Linear projection of the gradient onto Lp balls.

__call__(*grad: torch.Tensor*) → torch.Tensor

Process gradient with linear projection onto the Lp ball.

Sets the direction by maximizing the scalar product with the gradient over the Lp ball.

Parameters
grad (*torch.Tensor*) – Input gradients.

Returns
The gradient linearly projected onto the Lp ball.

Return type
torch.Tensor

Raises
NotImplementedError – Raises NotImplementedError if the norm is not in 2, inf.

__init__(*perturbation_model: str = 'l2'*) → None

Create linear projection for the gradient.

Parameters
perturbation_model (*str, optional*) – Perturbation model for the Lp ball, by default LpPerturbationModels.L2.

Raises
ValueError – Raises ValueError if the perturbation model is not implemented. Available, l1, l2, linf.

8.4 secmlt.optimization.initializer module

Initializers for the attacks.

class secmlt.optimization.initializer.**Initializer**

Bases: object

Initialization for the perturbation delta.

__call__(*x: torch.Tensor*) → torch.Tensor

Get initialization for the perturbation.

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

Returns
Initialized perturbation.

Return type
torch.Tensor

class secmlt.optimization.initializer.**RandomLpInitializer**(*radius: torch.Tensor*, *perturbation_model: LpPerturbationModels*)

Bases: *Initializer*

Random perturbation initialization in Lp ball.

__call__(*x: torch.Tensor*) → torch.Tensor

Get random perturbations.

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

Returns
Initialized random perturbations.

Return type
torch.Tensor

__init__(*radius: torch.Tensor*, *perturbation_model:* LpPerturbationModels) → None

Create random perturbation initializer.

Parameters

- **radius** (*torch.Tensor*) – Radius of the Lp ball for the constraint.
- **perturbation_model** (LpPerturbationModels) – Perturbation model for the constraint.

8.5 secmlt.optimization.optimizer_factory module

Optimizer creation tools.

class secmlt.optimization.optimizer_factory.**OptimizerFactory**

Bases: object

Creator class for optimizers.

OPTIMIZERS: ClassVar[dict[str, torch.optim.Optimizer]] = {'adam': torch.optim.Adam, 'sgd': torch.optim.SGD}

static create_adam(*lr: float*) → partial[torch.optim.Adam]

Create the Adam optimizer.

Parameters
lr (*float*) – Learning rate.

Returns
Adam optimizer.

Return type
functools.partial[Adam]

static create_from_name(*optimizer_name: str*, *lr: float*, ***kwargs*) → partial[torch.optim.Adam] | partial[torch.optim.SGD]

Create an optimizer.

Parameters

- **optimizer_name** (*str*) – One of the available optimizer names. Available: *adam*, *sgd*.
- **lr** (*float*) – Learning rate.

Returns
The created optimizer.

Return type
functools.partial[Adam] | functools.partial[SGD]

Raises
ValueError – Raises ValueError when the requested optimizer is not in the list of implemented optimizers.

static create_sgd(*lr: float*) → partial[torch.optim.SGD]

Create the SGD optimizer.

Parameters
lr (*float*) – Learning rate.

Returns
SGD optimizer.

Return type
functools.partial[SGD]

8.6 secmlt.optimization.random_perturb module

Random pertubations in Lp balls.

class secmlt.optimization.random_perturb.**RandomPerturb**(*p: str*, *epsilon: float*)

Bases: object

Random perturbation creator.

static __new__(*cls*, *p: str*, *epsilon: float*) → *RandomPerturbBase*

Creator for random perturbation in Lp norms.

Parameters

- **p** (*str*) – p-norm used for the random perturbation shape.
- **epsilon** (*float*) – Radius of the random perturbation constraint.

Returns
Random perturbation object.

Return type
RandomPerturbBase

Raises
ValueError – Raises ValueError if the norm is not in 0, 1, 2, inf.

class secmlt.optimization.random_perturb.**RandomPerturbBase**(*epsilon: float*)

Bases: ABC

Class implementing the random perturbations in Lp balls.

__call__(*x: torch.Tensor*) → torch.Tensor

Get the perturbations for the given samples.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

Returns
Perturbations (to apply) to the given samples.

Return type
torch.Tensor

__init__(*epsilon: float*) → None

Create random perturbation object.

Parameters
epsilon (*float*) – Constraint radius.

abstract get_perturb(*x: torch.Tensor*) → torch.Tensor

Generate random perturbation for the Lp norm.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

class secmlt.optimization.random_perturb.**RandomPerturbL0**(*epsilon: float*)

Bases: *RandomPerturbBase*

Random Perturbations for L0 norm.

get_perturb(*x: torch.Tensor*) → torch.Tensor

Generate random perturbation for the L0 norm.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

Returns
Perturbed samples.

Return type
torch.Tensor

class secmlt.optimization.random_perturb.**RandomPerturbL1**(*epsilon: float*)

Bases: *RandomPerturbBase*

Random Perturbations for L1 norm.

get_perturb(*x: torch.Tensor*) → torch.Tensor

Generate random perturbation for the L1 norm.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

Returns
Perturbed samples.

Return type
torch.Tensor

class secmlt.optimization.random_perturb.**RandomPerturbL2**(*epsilon: float*)

Bases: *RandomPerturbBase*

Random Perturbations for L2 norm.

get_perturb(*x: torch.Tensor*) → torch.Tensor

Generate random perturbation for the L2 norm.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

Returns
Perturbed samples.

Return type
torch.Tensor

class secmlt.optimization.random_perturb.**RandomPerturbLinf**(*epsilon: float*)

Bases: *RandomPerturbBase*

Random Perturbations for Linf norm.

get_perturb(*x: torch.Tensor*) → torch.Tensor

Generate random perturbation for the Linf norm.

Parameters
x (*torch.Tensor*) – Input samples to perturb.

Returns
Perturbed samples.

Return type
torch.Tensor

8.7 Module contents

Optimization functionalities.

CHAPTER

NINE

SECMLT.TRACKERS PACKAGE

9.1 Submodules

9.2 secmlt.trackers.image_trackers module

Image-specific trackers.

class secmlt.trackers.image_trackers.**GradientsTracker**

Bases: *Tracker*

Tracker for gradient images.

__init__() → None

Create gradients tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.Tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the gradients at the current iteration as images.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.image_trackers.**SampleTracker**

Bases: *Tracker*

Tracker for adversarial images.

__init__() → None

Create adversarial image tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.Tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the adversarial examples at the current iteration as images.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

9.3 secmlt.trackers.tensorboard_tracker module

Tensorboard tracking utilities.

class secmlt.trackers.tensorboard_tracker.**TensorboardTracker**(*logdir: str*, *trackers: list[Tracker] | None = None*)

Bases: *Tracker*

Tracker for Tensorboard. Uses other trackers as subscribers.

__init__(*logdir: str*, *trackers: list[Tracker] | None = None*) → None

Create tensorboard tracker.

Parameters

- **logdir** (*str*) – Folder to store tensorboard logs.
- **trackers** (*list[Tracker] | None, optional*) – List of trackers subsctibed to the updates, by default None.

get_last_tracked() → NotImplementedError

Not implemented for this tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Update all subscribed trackers.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

9.4 secmlt.trackers.trackers module

Trackers for attack metrics.

class secmlt.trackers.trackers.**GradientNormTracker**(*p:* LpPerturbationModels = *'l2'*)

Bases: *Tracker*

Tracker for gradients.

__init__(*p:* LpPerturbationModels = *'l2'*) → None

Create gradient norm tracker.

Parameters

p (LpPerturbationModels, *optional*) – Perturbation model to compute the norm, by default LpPerturbationModels.L2.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the sample-wise gradient of the loss w.r.t delta.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.trackers.**LossTracker**

Bases: *Tracker*

Tracker for attack loss.

__init__() → None

Create loss tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the sample-wise loss of the attack at the current iteration.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.trackers.**PerturbationNormTracker**(*p:* LpPerturbationModels = *'l2'*)

Bases: *Tracker*

Tracker for perturbation norm.

__init__(*p:* LpPerturbationModels = *'l2'*) → None

Create perturbation norm tracker.

Parameters

p (LpPerturbationModels, *optional*) – Perturbation model to compute the norm, by default LpPerturbationModels.L2.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the perturbation norm at the current iteration.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.trackers.**PredictionTracker**

Bases: *Tracker*

Tracker for model predictions.

__init__() → None

Create prediction tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the sample-wise model predictions at the current iteration.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.trackers.**ScoresTracker**(*y: int | torch.Tensor | None = None*)

Bases: *Tracker*

Tracker for model scores.

__init__(*y: int | torch.Tensor | None = None*) → None

Create scores tracker.

track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the sample-wise model scores at the current iteration.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

class secmlt.trackers.trackers.**Tracker**(*name: str*, *tracker_type: str = 'scalar'*)

Bases: ABC

Class implementing the trackers for the attacks.

__init__(*name: str*, *tracker_type: str = 'scalar'*) → None

Create tracker.

Parameters

- **name** (*str*) – Tracker name.
- **tracker_type** (*str, optional*) – Type of tracker (mostly used for tensorboard functionalities), by default SCALAR. Available: SCALAR, IMAGE, MULTI_SCALAR.

get() → torch.Tensor

Get the current tracking history.

Returns
History of tracked parameters.

Return type
torch.Tensor

get_last_tracked() → None | torch.Tensor

Get last element tracked.

Returns
Returns the last tracked element if anything was tracked.

Return type
None | torch.Tensor

abstract track(*iteration: int*, *loss: torch.Tensor*, *scores: torch.Tensor*, *x_adv: torch.tensor*, *delta: torch.Tensor*, *grad: torch.Tensor*) → None

Track the history of given attack observable parameters.

Parameters

- **iteration** (*int*) – The attack iteration number.
- **loss** (*torch.Tensor*) – The value of the (per-sample) loss of the attack.
- **scores** (*torch.Tensor*) – The output scores from the model.
- **x_adv** (*torch.tensor*) – The adversarial examples at the current iteration.
- **delta** (*torch.Tensor*) – The adversarial perturbations at the current iteration.
- **grad** (*torch.Tensor*) – The gradient of delta at the given iteration.

9.5 Module contents

Module implementing trackers for adversarial attacks.

CHAPTER

TEN

SECMLT.UTILS PACKAGE

10.1 Submodules

10.2 secmlt.utils.tensor_utils module

Basic utils for tensor handling.

secmlt.utils.tensor_utils.**atleast_kd**(*x: torch.Tensor*, *k: int*) → torch.Tensor

Add dimensions to the tensor x until it reaches k dimensions.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **k** (*int*) – Number of desired dimensions.

Returns

The input tensor x but with k dimensions.

Return type

torch.Tensor

10.3 Module contents

Utilities for the use of the library.

CHAPTER

ELEVEN

SECMLT: CONTRIBUTION GUIDE

SecMLT is an open-source Python library for Adversarial Machine Learning and robustness evaluation. We welcome contributions from the research community to expand its capabilities, improve its functionality, or add new features. In this guide, we will discuss how to contribute to SecMLT through forks, pull requests, and code formatting using Ruff.

11.1 Prerequisites

Before contributing to SecMLT:

1. Familiarize yourself with the library by reviewing the official documentation and exploring the existing codebase.
2. Install the required dependencies (refer to the installation guide).

11.2 Setting up your development environment

To contribute to SecMLT, follow these steps:

1. **Fork the repository**: Go to the SecMLT GitHub page and click "Fork" in the upper-right corner. This will create a copy of the SecMLT repository under your GitHub account.
2. **Clone your forked repository**: Clone your forked repository to your local machine using `git clone` command:

```
git clone <your_forked_repo_URL> secmlt
```

3. **Set up remote repositories**: Add the original SecMLT repository as an upstream remote and set the tracking branch to be `master`:

```
cd secmlt
git remote add upstream <original_repo_URL>
git fetch upstream
git checkout master --track upstream/master
```

11.3 Making changes

1. Create a new branch for your feature, bug fix, or documentation update:

```
git checkout -c <new_branch_name>
```

2. Make the necessary changes to the codebase (add features, fix bugs, improve documentation, etc.). Be sure to write clear and descriptive commit messages.
3. Test your changes locally using appropriate testing frameworks and tools.

11.4 Formatting your code

In our project, we leverage Ruff and pre-commit to enhance code quality and streamline the development process. Ruff is a static code linter, while Pre-commit is a framework for defining pre-commit hooks.

11.5 Documentation

When adding new functionalities, modules, or packages to SecMLT, it's essential to document them properly. This includes generating reStructuredText (RST) files, which are used by Sphinx to build the documentation.

To generate documentation RST files for new modules, follow these steps:

1. **Install the documentation requirements**: Ensure you have Sphinx and the docs dependencies installed. You can install it via pip:

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

2. **Write Docstrings**: Ensure your modules and functions/classes have docstrings following the reStructuredText format. This allows Sphinx to parse and generate documentation from your code. The following steps will find automatically the modules if they are properly documented.
3. **Run Autodoc**: Use the `sphinx-apidoc` command to automatically generate RST files for your modules:

```
sphinx-apidoc -o <output_directory> <module_directory> -f
```

   Replace `<output_directory>` with the directory where you want the RST files to be generated, and `<module_directory>` with the directory containing your modules. The `-f` parameter makes it rewrite existing files (so that the new functions are added).

   Specifically, it's easy to do it from the docs folder:

```
cd docs  # skip if you are already in the docs folder from the previous step
sphinx-apidoc -o ../docs/source ../src/secmlt -f
```

4. **Build Documentation**: Finally, build the documentation using Sphinx:

```
sphinx-build -b html <source_directory> <build_directory>
```

   Replace `<source_directory>` with the directory containing your Sphinx source files (including the generated RST files), and `<build_directory>` with the directory where you want the HTML documentation to be built.

   Specifically, you can use:

```
cd ..  # skip if you are already in the main project folder
sphinx-build -M html ./docs/source ./docs/build
```

By following these steps, you can ensure that your new modules are properly documented and integrated into the SecMLT documentation. A preview will be generated when you create the pull request.

11.5.1 Using Ruff

Ruff is integrated into our project to perform code linting. It helps ensure adherence to coding standards, identifies potential bugs, and enhances overall code quality. Here's how to use Ruff:

1. **Installation**: Make sure you have Ruff installed in your development environment. You can install it via pip:

   ```
   pip install ruff
   ```

2. **Running Ruff**: To analyze your codebase using Ruff, navigate to the project directory and run the following command:

   ```
   ruff check
   ```

   Ruff will analyze the codebase and provide feedback on potential issues and areas for improvement.

11.5.2 Using Pre-commit

Pre-commit is employed to automate various tasks such as code formatting, linting, and ensuring code consistency across different environments. We use it to enforce Ruff formatting *before* commit. Here's how to utilize Pre-commit:

1. **Installation**: Ensure that Pre-commit is installed in your environment. You can install it using pip:

   ```
   pip install pre-commit
   ```

2. **Configuration**: The project includes a `.pre-commit-config.yaml` file that specifies the hooks to be executed by Pre-commit. These hooks can include tasks such as code formatting, static analysis, and more.

3. **Installation of Hooks**: Run the following command in the project directory to install the Pre-commit hooks:

   ```
   pre-commit install
   ```

   This command will set up the hooks specified in the configuration file to run automatically before each commit.

4. **Running Pre-commit**: Whenever you make changes and attempt to commit them, Pre-commit will automatically execute the configured hooks. If any issues are found, Pre-commit will prevent the commit from proceeding and provide feedback on the detected issues.

11.5.3 Contributing with your Code

When contributing code to the project, follow these guidelines to ensure a smooth and efficient contribution process:

1. **Run Ruff and Pre-commit Locally**: Before making a pull request, run Ruff and Pre-commit locally to identify and fix potential issues in your code.

2. **Address Ruff and Pre-commit Warnings**: If Ruff or Pre-commit identifies any issues, address them before submitting your code for review. This ensures that the codebase maintains high standards of quality and consistency.

3. **Document Changes**: Clearly document any changes you make, including the rationale behind the changes and any potential impact on existing functionality.
4. If there are no issues with your code, commit the changes using the `git add` command and push them to your forked repository:

```
git add .
git commit -m "Your commit message"
git push origin <new_branch_name>
```

11.6 Submitting a pull request

1. Go to your forked repository on GitHub and click the "New pull request" button.
2. Choose the branch you've created as the source branch, and select `master` as the target branch.
3. Review the changes you're submitting and write a clear and descriptive pull request title and description.
4. Submit your pull request by clicking "Create pull request".
5. The SecMLT maintainers will review your pull request, provide feedback, or merge it into the main repository as appropriate.

We appreciate your contributions to SecMLT! If you have any questions or need assistance during the process, please don't hesitate to reach out to us on GitHub or other communication channels.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T