False Sense of Security

Release

Carlos Gomes

Nov 04, 2020

Chapter 1

Project

This work seeks to find convenient metrics that can be used to deduce the extent of gradient masking in an adversarially trained model.

Chapter 2

Documentation for the Code

`gradient_masking_tests.`**`get_accuracy`** (*model*, *test_dataset*, *attack=None*, *epsilon=0.03*, *subset_size=10000*, *device='cpu'*, *batch_size=32*)

Reports the accuracy of the model, potentially under some attack (e.g. FGSM, PGD, …)

`gradient_masking_tests.`**`run_masking_benchmarks`** (*model*, *test_dataset*, *epsilon=0.06*, *device='cpu'*, *batch_size=32*)

This method runs through a checklist of potential indicators of gradient masking, as exposed in "Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples" https://arxiv.org/pdf/1802.00420.pdf

`gradient_masking_tests.`**`spsa_accuracy`** (*model*, *test_dataset*, *eps=0.03*, *iters=1*, *nb_sample=128*, *batch_size=8*, *device='cpu'*, *subset_size=100*)

Reports the accuracy of the model under the SPSA attack. This method is quite expensive, so a small subset_size is reccomended, particularly for deeper networks.

class `Nets.`**`CIFAR_Net`** (*device='cpu'*)

Very simple Conv Net for CIFAR-10.

class `Nets.`**`CIFAR_Res_Net`** (*device*)

ResNet 18 from robustbench zoo extended with a normalization layer at the beggining.

class `Nets.`**`CIFAR_Wide_Res_Net`** (*device*)

WideResNet-28-10 from robustbench zoo(Carmon2019UnlabeledNet) extended with a normalization layer at the beggining.

class `Nets.`**`Gradient_Masked_MNIST`** (*device='cpu'*, *log_interval=100*, *batch_size=64*, *test_batch_size=1000*, *binary=False*, *eps=0.5*)

class `Nets.`**`MNIST_Net`** (*device='cpu'*, *log_interval=100*, *batch_size=64*, *test_batch_size=1000*, *oracle=None*, *binary=False*)

class `Nets.`**`Normalization`** (*device*, *mean*, *std*)

Torch layer that handles normalization of data.

This normalization class ensures the attacks do not need to know about the preprocessing steps done on the data, and therefore step size can remain unchaged.

class `Nets.`**`PGD_MNIST`** (*device='cpu'*, *log_interval=100*, *batch_size=64*, *test_batch_size=1000*, *step=0.1*, *eps=0.7*, *iters=7*, *binary=False*)

class `trainers.`**`CurvatureRegularizationTrainer`** (*device='cpu', log_interval=10, lambda_=4, report_gradient_norm=None*)

Extends the base trainer to train with curvature regularization FROM https://github.com/F-Salehi/CURE_robustness "Robustness via curvature regularization, and vice versa ", SM. Moosavi-Dezfooli, A. Fawzi, J. Uesato, and P. Frossard, CVPR 2019.

TODO: Currently not working.

`regularizer` (*model, criterion, inputs, targets, h=3.0*)

Regularizer term in CURE

class `trainers.`**`FGSMTrainer`** (*device='cpu', log_interval=10, clip_min=0, clip_max=1, eps=0.03137254901960784, report_gradient_norm=None*)

Extends base trainer class to implement training with a single FGSM step.

class `trainers.`**`GradientRegularizationTrainer`** (*device='cpu', log_interval=10, lambda_=0.1, annealing=False, report_gradient_norm=None*)

Extends the base trainer class to implement training with input gradient regularization.

TODO: Decide on good value for lambda, good annealing schedule.

class `trainers.`**`Trainer`** (*device='cpu', log_interval=10, report_gradient_norm=None*)

Base class that trains a given model, given a dataloader using standard SGD.

All other trainer classes are extended from this one.

`utils.`**`adversarial_accuracy`** (*model, dataset_loader, attack=<function pgd_>, iters=20, eps=0.5, step=0.1, random_step=False, device='cpu'*)

Compute the adversarial accuracy of a model.

Deprecated, moved to using foolbox, advertorch.

`utils.`**`fgsm_`** (*model, x, target, eps=0.5, targeted=True, device='cpu', clip_min=None, clip_max=None, **kwargs*)

Internal process for all FGSM and PGD attacks used during training.

Returns the adversarial examples crafted from the inputs using the FGSM attack.

`utils.`**`gradient_information`** (*model, data, target, step=0.01, eps=0.5, iters=20, targeted=False, device='cpu', clip_min=None, clip_max=None*)

Computes the cosine information between the gradient of point at the decision boundary w.r.t. the loss and the vector (point at decision boundary - original input point).

For non gradient masked models, this point should be the closest one to the input that is at the decision boundary. Thus, we would expect these vectors to be +- collinear.

TODO: Incorrect implementation!!!!!! Do not use PGD, use something like deepfool. And do not use loss function for decision boundary, but rather difference in logits. TODO: Move to metrics

`utils.`**`gradient_norm`** (*model, data_loader, device='cpu'*)

Computes the gradient norm w.r.t. the loss at the given points.

TODO: Move to metrics.

`utils.`**`pgd_`** (*model, x, target, step, eps, iters=7, targeted=True, device='cpu', clip_min=None, clip_max=None, random_step=True, early_stop=False*)

Internal pgd attack used during training.

Applies iterated steps of the fgsm attack, projecting back to the relevant domain after each step.

- *Index*
- *Module Index*

- *Search Page*

Python Module Index