robustness

Release 1.0

Mar 04, 2021

Contents

Install via `pip`: `pip install robustness`

`robustness` is a package we (students in the MadryLab) created to make training, evaluating, and exploring neural networks flexible and easy. We use it in almost all of our projects (whether they involve adversarial training or not!) and it will be a dependency in many of our upcoming code releases. A few projects using the library include:

- Code for "Learning Perceptually-Aligned Representations via Adversarial Robustness" [EIS+19]
- Code for "Image Synthesis with a Single (Robust) Classifier" [STE+19]
- Code for "BREEDS: Benchmarks for Subpopulation Shift" [STM20]

We demonstrate how to use the library in a set of walkthroughs and our API reference. Functionality provided by the library includes:

- Training and evaluating standard and robust models for a variety of datasets/architectures using a *CLI interface*. The library also provides support for adding *custom datasets* and *model architectures*.

```
python -m robustness.main --dataset cifar --data /path/to/cifar \
   --adv-train 0 --arch resnet18 --out-dir /logs/checkpoints/dir/
```

- Performing *input manipulation* using robust (or standard) models—this includes making adversarial examples, inverting representations, feature visualization, etc. The library offers a variety of optimization options (e.g. choice between real/estimated gradients, Fourier/pixel basis, custom loss functions etc.), and is easily extendable.

```
import torch as ch
from robustness.datasets import CIFAR
from robustness.model_utils import make_and_restore_model

ds = CIFAR('/path/to/cifar')
model, _ = make_and_restore_model(arch='resnet50', dataset=ds,
             resume_path='/path/to/model', state_dict_path='model')
model.eval()
attack_kwargs = {
   'constraint': 'inf', # L-inf PGD
   'eps': 0.05, # Epsilon constraint (L-inf norm)
   'step_size': 0.01, # Learning rate for PGD
   'iterations': 100, # Number of PGD steps
   'targeted': True # Targeted attack
   'custom_loss': None # Use default cross-entropy loss
}

_, test_loader = ds.make_loaders(workers=0, batch_size=10)
im, label = next(iter(test_loader))
target_label = (label + ch.randint_like(label, high=9)) % 10
adv_out, adv_im = model(im, target_label, make_adv, **attack_kwargs)
```

- Importing `robustness` as a package, which allows for easy training of neural networks with support for custom loss functions, logging, data loading, and more! A good introduction can be found in our two-part walkthrough (*Part 1*, *Part 2*).

```
from robustness import model_utils, datasets, train, defaults
from robustness.datasets import CIFAR

# We use cox (http://github.com/MadryLab/cox) to log, store and analyze
# results. Read more at https//cox.readthedocs.io.
from cox.utils import Parameters
import cox.store
```

(continues on next page)

(continued from previous page)

```
# Hard-coded dataset, architecture, batch size, workers
ds = CIFAR('/path/to/cifar')
m, _ = model_utils.make_and_restore_model(arch='resnet50', dataset=ds)
train_loader, val_loader = ds.make_loaders(batch_size=128, workers=8)

# Create a cox store for logging
out_store = cox.store.Store(OUT_DIR)

# Hard-coded base parameters
train_kwargs = {
    'out_dir': "train_out",
    'adv_train': 1,
    'constraint': '2',
    'eps': 0.5,
    'attack_lr': 1.5,
    'attack_steps': 20
}
train_args = Parameters(train_kwargs)

# Fill whatever parameters are missing from the defaults
train_args = defaults.check_and_fill_args(train_args,
                        defaults.TRAINING_ARGS, CIFAR)
train_args = defaults.check_and_fill_args(train_args,
                        defaults.PGD_ARGS, CIFAR)

# Train a model
train.train_model(train_args, m, (train_loader, val_loader), store=out_store)
```

CHAPTER 1

Citation

If you use this library in your research, cite it as follows:

```
@misc{robustness,
   title={Robustness (Python Library)},
   author={Logan Engstrom and Andrew Ilyas and Shibani Santurkar and Dimitris Tsipras}
,
   year={2019},
   url={https://github.com/MadryLab/robustness}
}
```

(Have you used the package and found it useful? Let us know!).

CHAPTER 2

Walkthroughs

2.1 Training and evaluating networks via command line

In this walkthrough, we'll go over how to train and evaluate networks via the *robustness.main* command-line tool.

2.1.1 Training a standard (nonrobust) model

We'll start by training a standard (non-robust) model. This is accomplished through the following command:

```
python -m robustness.main --dataset DATASET --data /path/to/dataset \
   --adv-train 0 --arch ARCH --out-dir /logs/checkpoints/dir/
```

In the above, `DATASET` can be any supported dataset (i.e. in *robustness.datasets.DATASETS*). For a demonstration of how to add a supported dataset, see *here*.

With the above command, you should start seeing progress bars indicating that the training has begun! Note that there are a whole host of arguments that you can customize in training, including optimizer parameters (e.g. `--lr`, `--weight-decay`, `--momentum`), logging parameters (e.g. `--log-iters`, `--save-ckpt-iters`), and learning rate schedule. To see more about these arguments, we run:

```
python -m robustness --help
```

For completeness, the full list of parameters related to *non-robust* training are below:

```
--out-dir OUT_DIR     where to save training logs and checkpoints (default:
                    required)
                      config path for loading in parameters (default: None)
--exp-name EXP_NAME   where to save in (inside out_dir) (default: None)
--dataset {imagenet,restricted_imagenet,cifar,cinic,a2b}
                      (choices: {arg_type}, default: required)
--data DATA           path to the dataset (default: /tmp/)
--arch ARCH           architecture (see {cifar,imagenet}_models/ (default:
```

(continues on next page)

(continued from previous page)

```
                      required)
--batch-size BATCH_SIZE
                      batch size for data loading (default: by dataset)
--workers WORKERS     data loading workers (default: 30)
--resume RESUME       path to checkpoint to resume from (default: None)
--data-aug {0,1}      whether to use data augmentation (choices: {arg_type},
                      default: 1)
--epochs EPOCHS       number of epochs to train for (default: by dataset)
--lr LR               initial learning rate for training (default: 0.1)
--weight_decay WEIGHT_DECAY
                      SGD weight decay parameter (default: by dataset)
--momentum MOMENTUM   SGD momentum parameter (default: 0.9)
--step-lr STEP_LR     number of steps between 10x LR drops (default: by
                      dataset)
--step-lr-gamma GAMMA multiplier for each LR drop (default: 0.1, i.e., 10x drops)
--custom-lr-multiplier CUSTOM_SCHEDULE
                      LR sched (format: [(epoch, LR),...]) (default: None)
--lr-interpolation {linear, step}
                      How to interpolate between learning rates (default: step)
--log-iters LOG_ITERS
                      how frequently (in epochs) to log (default: 5)
--save-ckpt-iters SAVE_CKPT_ITERS
                      how frequently (epochs) to save (-1 for bash, only
                      saves best and last) (default: -1)
--mixed-precision {0, 1}
                      Whether to use mixed-precision training (needs
                      to be compiled with NVIDIA AMP support)
```

Finally, there is one additional argument, `--adv-eval 0,1`, that enables adversarial evaluation of the non-robust model as it is being trained (i.e. instead of reporting just standard accuracy every few epochs, we'll also report robust accuracy if `--adv-eval 1` is added). However, adding this argument also necessitates the addition of hyperparameters for adversarial attack, which we cover in the following section.

2.1.2 Training a robust model (adversarial training)

To train a robust model we proceed in the exact same way as for a standard model, but with a few changes. First, we change `--adv-train 0` to `--adv-train 1` in the training command. Then, we need to make sure to supply all the necessary hyperparameters for the attack:

```
--attack-steps ATTACK_STEPS
                    number of steps for adversarial attack (default: 7)
--constraint {inf,2,unconstrained}
                      adv constraint (choices: {arg_type}, default:
                      required)
--eps EPS             adversarial perturbation budget (default: required)
--attack-lr ATTACK_LR
                      step size for PGD (default: required)
--use-best {0,1}      if 1 (0) use best (final) PGD step as example
                      (choices: {arg_type}, default: 1)
--random-restarts RANDOM_RESTARTS
                      number of random PGD restarts for eval (default: 0)
--custom-eps-multiplier EPS_SCHEDULE
                      epsilon multiplier sched (same format as LR schedule)
```

2.1.3 Evaluating trained models

To evaluate a trained model, we use the `--eval-only` flag when calling *robustness.main*. To evaluate the model for just standard (not adversarial) accuracy, only the following arguments are required:

```
python -m robustness.main --dataset DATASET --data /path/to/dataset \
   --eval-only 1 --out-dir OUT_DIR --arch arch --adv-eval 0 \
   --resume PATH_TO_TRAINED_MODEL_CHECKPOINT
```

We can also evaluate adversarial accuracy by changing `--adv-eval 0` to `--adv-eval 1` and also adding the arguments from the previous section used for adversarial attacks.

2.1.4 Examples

Training a non-robust ResNet-18 for the CIFAR dataset:

```
python -m robustness.main --dataset cifar --data /path/to/cifar \
   --adv-train 0 --arch resnet18 --out-dir /logs/checkpoints/dir/
```

Training a robust ResNet-50 for the Restricted-ImageNet dataset:

```
CUDA_VISIBLE_DEVICES=1,2,3,4,5,6 python -m robustness.main --dataset restricted_
↪imagenet --data \
   $IMAGENET_PATH --adv-train 1 --arch resnet50 \
   --out-dir /tmp/logs/checkpoints/dir/ --eps 3.0 --attack-lr 0.5 \
   --attack-steps 7 --constraint 2
```

Testing the standard and adversarial accuracy of a trained CIFAR-10 model with L2 norm constraint of 0.5 and 100 L2-PGD steps:

```
python -m robustness.main --dataset cifar --eval-only 1 --out-dir /tmp/ \
--arch resnet50 --adv-eval 1 --constraint 2 --eps 0.5 --attack-lr 0.1 \
--attack-steps 100 --resume path/to/ckpt/checkpoint.pt.best
```

2.1.5 Reading and analyzing training results

By default, the above command will store all the data generated from the training process above in a subdirectory inside of `/logs/checkpoints/dir/`, the path supplied to the `--out-dir` argument. The subdirectory will be named by default via a 36 character, randomly generated unique identifier, but it can be named manually via the `--exp-name` argument. By the end of training, the folder structure will look something like like:

```
/logs/checkpoints/dir/a9ffc412-595d-4f8c-8e35-41f000cd35ed
   checkpoint.latest.pt
   checkpoint.best.pt
   store.h5
   tensorboard/
   save/
```

This is the file structure of a data store from the Cox logging library. It contains all the tables (stored as Pandas dataframes, in HDF5 format) of data we wrote about the experiment:

```
>>> from cox import store
>>> s = store.Store('/logs/checkpoints/dir/', '6aeae7de-3549-49d5-adb6-52fe04689b4e')
>>> s.tables
{'ckpts': <cox.store.Table object at 0x7f09a6ae99b0>, 'logs': <cox.store.Table object␣
↪at 0x7f09a6ae9e80>, 'metadata': <cox.store.Table object at 0x7f09a6ae9dd8>}
```

We can get the metadata by looking at the metadata table and extracting values we want. For example, if we wanted to get the learning rate, 0.1:

```
>>> s['metadata'].df['lr']
0    0.1
Name: lr, dtype: float64
```

Or, if we wanted to find out which epoch had the highest validation accuracy:

```
>>> l_df = s['logs']
>>> ldf[ldf['nat_prec1'] == max(ldf['nat_prec1'].tolist())]['epoch'].tolist()[0]
32
```

In a similar manner, the 'ckpts' table contains all the previous checkpoints, and the 'logs' table contains logging information pertaining to the training. Cox allows us to really easily aggregate training logs across different training runs and compare/analyze them—we recommend taking a look at the Cox documentation for more information on how to use it.

Note that when training models programmatically (as in our walkthrough *Part 1* and *Part 2*), it is possible to add on custom logging functionalities and keep track of essentially anything during training.

2.2 Input manipulation with pre-trained models

The robustness library provides functionality to perform various input space manipulations using a trained model. This ranges from basic manipulation such as creating untargeted and targeted adversarial examples, to more advanced/custom ones. These types of manipulations are precisely what we use in the code releases for our papers [EIS+19] and [STE+19].

2.2.1 Generating Untargeted Adversarial Examples

First, we will go through an example of how to create untargeted adversarial examples for a ResNet-50 on the dataset CIFAR-10.

1. Load the dataset using:

   ```
   import torch as ch
   from robustness.datasets import CIFAR
   ds = CIFAR('/path/to/cifar')
   ```

2. Restore the pre-trained model using:

   ```
   from robustness.model_utils import make_and_restore_model
   model, _ = make_and_restore_model(arch='resnet50', dataset=ds,
                resume_path=PATH_TO_MODEL)
   model.eval()
   ```

3. Create a loader to iterate through the test set (set hyperparameters appropriately). Get a batch of test image-label pairs:

```
_, test_loader = ds.make_loaders(workers=NUM_WORKERS,
                                 batch_size=BATCH_SIZE)
_, (im, label) = next(enumerate(test_loader))
```

4. Define attack parameters (see *robustness.attacker.AttackerModel.forward()* for documentation on these parameters):

```
kwargs = {
    'constraint':'2', # use L2-PGD
    'eps': ATTACK_EPS, # L2 radius around original image
    'step_size': ATTACK_STEPSIZE,
    'iterations': ATTACK_STEPS,
    'do_tqdm': True,
}
```

   The example considers a standard PGD attack on the cross-entropy loss. The parameters that you might want to customize are `constraint` (`'2'` and `'inf'` are supported), `'attack_eps'`, `'step_size'` and `'iterations'`.

5. Find adversarial examples for `im` from step 3:

```
_, im_adv = model(im, label, make_adv=True, **kwargs)
```

6. If we want to visualize adversarial examples `im_adv` from step 5, we can use the utilities provided in the *robustness.tools* module:

```
from robustness.tools.vis_tools import show_image_row
from robustness.tools.label_maps import CLASS_DICT

# Get predicted labels for adversarial examples
pred, _ = model(im_adv)
label_pred = ch.argmax(pred, dim=1)

# Visualize test set images, along with corresponding adversarial examples
show_image_row([im.cpu(), im_adv.cpu()],
         tlist=[[CLASS_DICT['CIFAR'][int(t)] for t in l] for l in [label, label_
↪pred]],
         fontsize=18,
         filename='./adversarial_example_CIFAR.png')
```

Here is a sample output visualization snippet from step 6:

Fig. 1: Random samples from the CIFAR-10 test set (top row), along with their corresponding untargeted adversarial examples (bottom row). Image titles correspond to ground truth labels and predicted labels for the top and bottom row respectively.

2.2.2 Generating Targeted Adversarial Examples

The procedure for creating untargeted and targeted adversarial examples using the robustness library are very similar. In fact, we will start by repeating steps 1-3 describe above. The rest of the procedure is as follows (most of it involves minor modifications to steps 4-5 above):

1. Define attack parameters:

```
kwargs = {
    'constraint':'2',
    'eps': ATTACK_EPS,
    'step_size': ATTACK_STEPSIZE,
    'iterations': ATTACK_STEPS,
    'targeted': True,
    'do_tqdm': True
}
```

   The key difference from step 4 above is the inclusion of an additional parameter `'targeted'` which is set to `True` in this case.

2. Define target classes towards which we want to perturb `im`. For instance, we can perturb all the images towards class `0`:

```
targ = ch.zeros_like(label)
```

3. Find adversarial examples for `im`:

```
_, im_adv = model(im, targ, make_adv=True, **kwargs)
```

If you would like to visualize the targeted adversarial examples, you can repeat the aforementioned step 6, i.e. the *`robustness.tools.vis_tools.show_image_row()`* method:

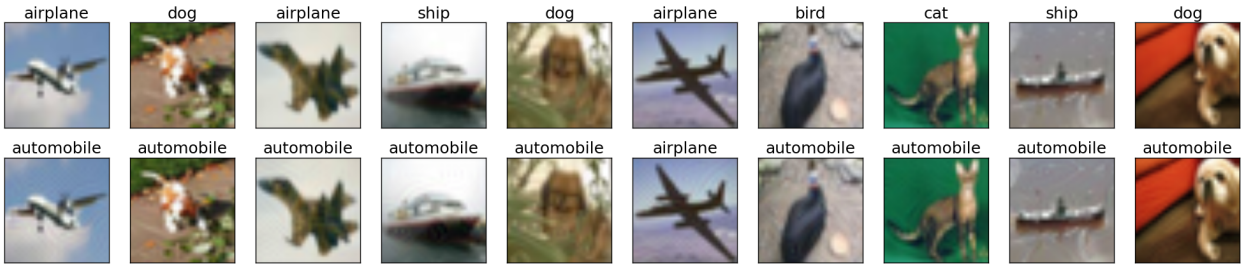


Fig. 2: Random samples from the CIFAR-10 test set (top row), along with their corresponding targeted adversarial examples (bottom row). Image titles correspond to ground truth labels and predicted labels (target labels) for the top and bottom row respectively.

2.2.3 Custom Input Manipulation (e.g. Representation Inversion)

You can also use the robustness lib functionality to perform input manipulations beyond adversarial attacks. In order to do this, you will need to define a custom loss function (to replace the default `ch.nn.CrossEntropyLoss`).

We will now walk through an example of defining a custom loss for inverting the representation (output of the pre-final network layer, before the linear classifier) for a given image. Specifically, given the representation for an image, our goal is to find (starting from noise) an input whose representation is close by (in terms of euclidean distance).

First, we will repeat steps 1-3 from the procedure for generating untargeted adversarial examples.

1. Load a set of images to invert and find their representation. Here we choose random samples from the test set:

```
_, (im_inv, label_inv) = next(enumerate(test_loader)) # Images to invert
with ch.no_grad():
    (_, rep_inv), _ = model(im_inv, with_latent=True) # Corresponding␣
↪representation
```

2. We now define a custom loss function that penalizes difference from a target representation `targ`:

```
def inversion_loss(model, inp, targ):
   # Compute representation for the input
   _, rep = model(inp, with_latent=True, fake_relu=True)
   # Normalized L2 error w.r.t. the target representation
   loss = ch.div(ch.norm(rep - targ, dim=1), ch.norm(targ, dim=1))
   return loss, None
```

3. We are now ready to define the attack args :samp: *kwargs*. This time we will supply our custom loss `inversion_loss`:

```
kwargs = {
    'custom_loss': inversion_loss,
    'constraint':'2',
    'eps': 1000,
    'step_size': 1,
    'iterations': 10000,
    'targeted': True,
    'do_tqdm': True,
}
```

4. We now define a seed input which will be the starting point for our inversion process. We will just use a gray image with (scaled) Gaussian noise:

```
im_seed = ch.clamp(ch.randn_like(im_inv) / 20 + 0.5, 0, 1)
```

5. Finally, we are ready to perform the inversion:

```
_, im_matched = model(im_seed, rep_inv, make_adv=True, **kwargs)
```

6. We can also visualize the results of the inversion process (similar to step 6 above)::

```
show_image_row([im_inv.cpu(), im_seed.cpu(), im_matched.cpu()],
         ["Original", r"Seed ($x_0$)", "Result"],
         fontsize=18,
         filename="./custom_inversion_CIFAR.png")
```

You should see something like this:

Changing optimization methods

In the above, we consistently used L2-PGD for all of our optimization tasks, as dictated by the value of the `constraint` argument to the model. However, there are a few more possible optimization methods available in the `robustness` package by default. We give a brief overview of them here—the *robustness.attack_steps* module has more in-depth documentation on each method:

- If `kwargs['constraint'] == '2'` (as it was for this walkthrough), then `eps` is interpreted as an L2-norm constraint, and we take ℓ_2-normalized gradient steps. More information at *robustness.attack_steps.L2Step*.

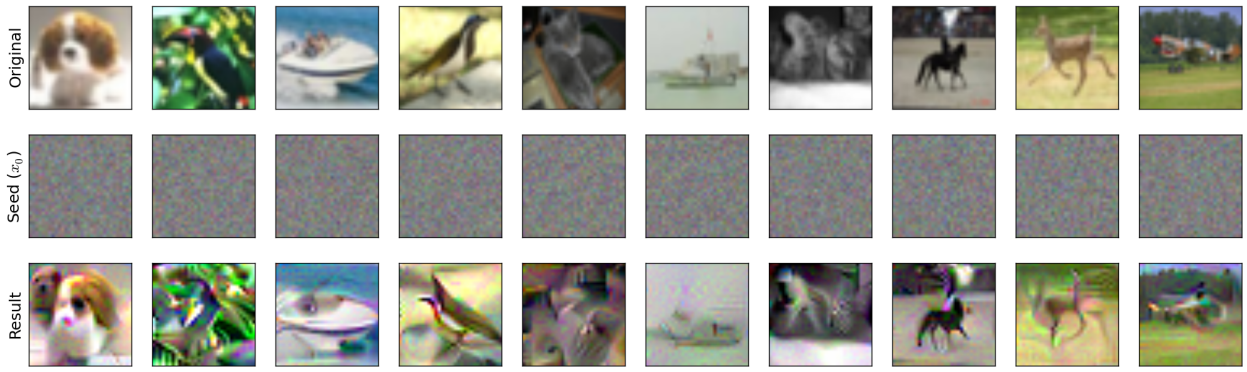


Fig. 3: Inverting representations of a robust network. Starting from the seed (middle row), we optimize for an image (bottom row) that is close to the representation of the original image (top row).

- If `kwargs['constraint'] == 'inf'`, then `eps` is interpreted as an ℓ_∞ norm constraint, and we take ℓ_∞-normalized PGD steps (i.e. signed gradient steps). More information at *`robustness.attack_steps.LinfStep`*.
- If `kwargs['constraint'] == 'unconstrained'`, then `eps` is ignored and the adversarial image is only clipped to the [0, 1] range. The optimization method is ordinary gradient descent, i.e., no projection is done to the gradient before making a step.
- If `kwargs['constraint'] == 'fourier'`, then `eps` is ignored and the adversarial image is only clipped to the [0, 1] range. The optimization method is once again ordinary gradient descent, but this time in the Fourier basis rather than the pixel basis. This tends to yield nicer-looking images, for reasons discussed here [OMS17].

For example, in order to do representation inversion in the fourier basis rather than the pixel basis, we would change `kwargs` as follows:

```
kwargs = {
    'custom_loss': inversion_loss,
    'constraint':'fourier',
    'eps': 1000, # ignored anyways
    'step_size':  5000, # have to re-tune LR
    'iterations': 1000,
    'targeted': True,
    'do_tqdm': True,
}
```

We also have to change our `im_seed` to be a valid Fourier-basis parameterization of an image:

```
im_seed = ch.randn(BATCH_SIZE, 3, 32, 32, 2) / 5 # Parameterizes a grey image
```

Running this through the model just like before and visualizing the results should yield something like:

```
show_image_row([im_inv.cpu(), im_matched.cpu()],
      ["Original", "Result"],
      fontsize=18,
      filename='./custom_inversion_CIFAR_fourier.png')
```

Below we show how to implement our own custom optimization methods as well.

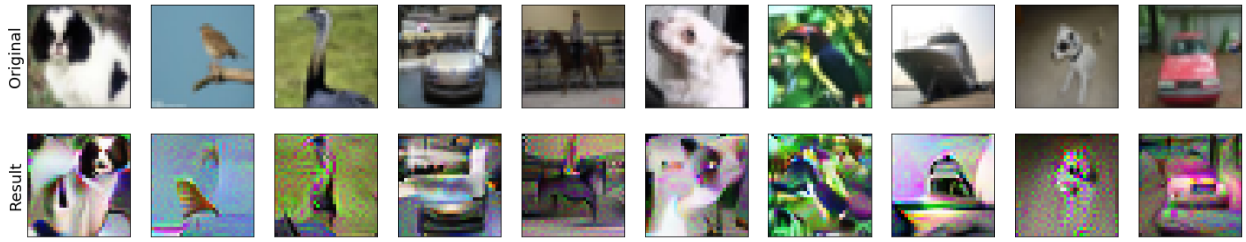


Fig. 4: Inverting representations of a robust network in the fourier basis. Starting from the seed (middle row), we optimize for an image (bottom row) that is close to the representation of the original image (top row).

2.2.4 Advanced usage

The steps given above are sufficient for using the `robustness` library for a wide a variety of input manipulation techniques, and for reproducing the code for our papers. Here we go through a couple more advanced options and techniques for using the library.

Gradient Estimation/NES

The first more advanced option we look at is how to use estimated gradients in place of real ones when doing adversarial attacks (this corresponds to the NES black-box attack [IEA+18]). To do this, we simply need to provide the `est_grad` argument (`None` by default) to the model upon creation of the adversarial example. As discussed in *this docstring*, the proper format for this argument is a tuple of the form `(R, N)`, and the resulting gradient estimator is

$$\nabla_x f(x) \approx \sum_{i=0}^{N} f(x + R \cdot \vec{\delta_i}) \cdot \vec{\delta_i},$$

where δ_i are randomly sampled from the unit ball (note that we employ antithetic sampling to reduce variance, meaning that we actually draw $N/2$ random vectors from the unit ball, and then use $\delta_{N/2+i} = -\delta_i$.

To do representation inversion with NES/estimated gradients, we would just need to add the following line to the above:

```
kwargs['est_grad'] = (0.5, 100)
```

This will run the optimization process using 100-query gradient estimates with $R = 0.5$ in the above estimator.

Custom optimization methods

To make a custom optimization method, all that is required is to subclass *robustness.attack_steps.AttackerStep* class. The class documentation has information about which methods and properties need to be implemented, but perhaps the most illustrative thing is an example of how to implement the ℓ_2-PGD attacker:

```
class L2Step(AttackerStep):
    def project(self, x):
        diff = x - self.orig_input
        diff = diff.renorm(p=2, dim=0, maxnorm=self.eps)
        return ch.clamp(self.orig_input + diff, 0, 1)

    def step(self, x, g):
        # Scale g so that each element of the batch is at least norm 1
```

(continues on next page)

(continued from previous page)

```
        l = len(x.shape) - 1
        g_norm = ch.norm(g.view(g.shape[0], -1), dim=1).view(-1, *([1]*l))
        scaled_g = g / (g_norm + 1e-10)
        return x + scaled_g * self.step_size

    def random_perturb(self, x):
        new_x = x + (ch.rand_like(x) - 0.5).renorm(p=2, dim=1, maxnorm=self.eps)
        return ch.clamp(new_x, 0, 1)

    def to_image(self, x):
        return x
```

As part of initialization, the properties `self.orig_input`, `self.eps`, and `self.step_size`, and `self.use_grad` will already be defined. As shown above, there are four functions that your custom step can implement (three of which, `project`, `step`, and `random_perturb`, are mandatory).

- The `project` function takes in an input `x` and projects it to the appropriate ball around `self.orig_input`.
- The `step` function takes in a current input and a gradient, and outputs the next iterate of the optimization process (in our case, we normalize the gradient and then add it to the iterate to perform an ℓ_2 projected gradient ascent step).
- The `random_perturb` function is called if the `random_start` option is given when constructing the adversarial example. Given an input, it should apply a small random perturbation to the input while still keeping it within the adversarial budget.
- The `to_image` function is identity by default, but is useful when working with alternative parameterizations of images. For example, when optimizing in Fourier space, the `to_image` function takes in an input which is in the Fourier basis, and outputs a valid image.

2.3 Using robustness as a general training library (Part 1: Getting started)

In the other walkthroughs, we've demonstrated how to use `robustness` as a *command line tool for training and evaluating models*, and how to use it as a library for *input manipulation*. Here, we'll demonstrate how `robustness` can be used a general library for experimenting with neural network training. We've found the library has saved us a tremendous amount of time both writing boilerplate code and making custom modifications to the training process (one of the primary goals of the library is to make such modifications simple).

This walkthrough will be split into two parts: in the first part (this one), we'll show how to get started with the `robustness` library, and go through the process of making a `main.py` file for training neural networks. In the *second part*, we'll show how to customize the training process through custom loss functions, architectures, datasets, logging, and more.

2.3.1 Step 1: Imports

Our goal in this tutorial will be to make a python file that works nearly identically to the robustness *Command-line tool*. That is, a user will be able to call `python main.py [--arg value ...]` to train a standard or robust model. We'll start by importing the necessary modules from the package:

```
from robustness.datasets import DATASETS
from robustness.model_utils import make_and_restore_model
from robustness.train import train_model
```

(continues on next page)

(continued from previous page)

```
from robustness.defaults import check_and_fill_args
from robustness.tools import constants, helpers
from robustness import defaults
```

To make life easier, we use cox (a super lightweight python logging library) for logging:

```
from cox import utils
from cox import store
```

Finally, we'll also need the following external imports:

```
import torch as ch
from argparse import ArgumentParser
import os
```

2.3.2 Step 2: Dealing with arguments

In this first step, we'll set up an `args` object which has all the parameters we need to train our model. In Step 2.1 we'll show how to use `argparse` to accept user input for specifying parameters via command line; in Step 2.2 we show how to sanity-check the `args` object and fill in reasonable defaults.

Step 2.1: Setting up command-line args

The first real step in making our main file is setting up an `argparse.ArgumentParser` object to parse command-line arguments for us. (If you are not familiar with the python argparses module, we recommend looking there first). Note that if you're not interested in accepting command-line input for arguments via argparse, you can skip to Step 2.2.

The `robustness` package provides the *`robustness.defaults`* module to make dealing with arguments less painful. In particular, the properties *`robustness.defaults.TRAINING_ARGS`*, *`robustness.defaults.PGD_ARGS`*, and *`robustness.defaults.MODEL_LOADER_ARGS`*, contain all of the arguments (along with default values) needed for training models:

- *`TRAINING_ARGS`* has all of the model training arguments, like learning rate, momentum, weight decay, learning rate schedule, etc.
- *`PGD_ARGS`* has all of the arguments needed only for adversarial training, like number of PGD steps, perturbation budget, type of norm constraint, etc.
- *`MODEL_LOADER_ARGS`* has all of the arguments for instantiating the model and data loaders: dataset, path to dataset, batch size, number of workers, etc.

You can take a look at the documentation of *`robustness.defaults`* to learn more about how these attributes are set up and see exactly which arguments they contain and with what defaults, as well as which arguments are required. The important thing is that the `robustness` package provides the function *`robustness.defaults.add_args_to_parser()`* which takes in an arguments object like the above, and an `argparse` parser, and adds the arguments to the parser:

```
parser = ArgumentParser()
parser = defaults.add_args_to_parser(defaults.MODEL_LOADER_ARGS, parser)
parser = defaults.add_args_to_parser(defaults.TRAINING_ARGS, parser)
parser = defaults.add_args_to_parser(defaults.PGD_ARGS, parser)
# Note that we can add whatever extra arguments we want to the parser here
args = parser.parse_args()
```

Important note: Even though the arguments objects do specify defaults for the arguments, these defaults are **not** given to the parser directly. More on this in Step 2.2.

If you don't want to use `argparse` and already know the values you want to use for the parameters, you can look at the *robustness.defaults* documentation, and hard-code the desired arguments as follows:

```
# Hard-coded base parameters
train_kwargs = {
    'out_dir': "train_out",
    'adv_train': 1,
    'constraint': '2',
    'eps': 0.5,
    'attack_lr': 1.5,
    'attack_steps': 20
}

# utils.Parameters is just an object wrapper for dicts implementing
# getattr and settattr
train_args = utils.Parameters(train_kwargs)
```

Step 2.2: Sanity checks and defaults

We generally found the `ArgumentParser` defaults to be too restrictive for our applications, and we also wanted to be able to fill in argument defaults even when we were not using `ArgumentParser`. Thus, we fill in default arguments separately via the *robustness.defaults.check_and_fill_args()* function. This function takes in the `args` Namespace object (basically anything exposing `setattr` and `getattr` functions), the same `ARGS` attributes discussed above, and a dataset class (used for filling in per-dataset defaults). The function fills in the defaults it has, and then throws an error if a required argument is missing:

```
assert args.dataset is not None, "Must provide a dataset"
ds_class = DATASETS[args.dataset]

args = check_and_fill_args(args, defaults.TRAINING_ARGS, ds_class)
if args.adv_train or args.adv_eval:
  args = check_and_fill_args(args, defaults.PGD_ARGS, ds_class)
args = check_and_fill_args(args, defaults.MODEL_LOADER_ARGS, ds_class)
```

Note that the *check_and_fill_args()* function is totally independent of `argparse`, and can be used even when you don't want to support command-line arguments. It's a really useful way of sanity checking the `args` object to make sure that there aren't any missing or misspecified arguments.

2.3.3 Step 3: Creating the model, dataset, and loader

The next step is to create the model, dataset, and data loader. We start by creating the dataset and loaders as follows:

```
# Load up the dataset
data_path = os.path.expandvars(args.data)
dataset = DATASETS[args.dataset](data_path)

# Make the data loaders
train_loader, val_loader = dataset.make_loaders(args.workers,
                args.batch_size, data_aug=bool(args.data_aug))

# Prefetches data to improve performance
```

(continues on next page)

(continued from previous page)

```
train_loader = helpers.DataPrefetcher(train_loader)
val_loader = helpers.DataPrefetcher(val_loader)
```

We can now create the model by using the *robustness.model_utils.make_and_restore_model()* function. This function is used for both creating new models, or (if a `resume_path` if passed) loading previously saved models.

```
model, _ = make_and_restore_model(arch=args.arch, dataset=dataset)
```

2.3.4 Step 4: Training the model

Finally, we create a cox Store for saving the results of the training, and call *robustness.train.train_model()* to begin training:

```
# Create the cox store, and save the arguments in a table
store = store.Store(args.out_dir, args.exp_name)
args_dict = args.as_dict() if isinstance(args, utils.Parameters) else vars(args)
schema = store.schema_from_dict(args_dict)
store.add_table('metadata', schema)
store['metadata'].append_row(args_dict)

model = train_model(args, model, (train_loader, val_loader), store=store)
```

2.4 Using robustness as a general training library (Part 2: Customizing training)

In this document, we'll continue our walk through using robustness as a library. In the *first part*, we made a `main.py` file that trains a model given user-specified parameters. For this part of the walkthrough, we'll continue from that `main.py` file. You can also start with a copy the source of *robustness.main*, or (if you don't want the full flexibility of all of those arguments) the following bare-bones `main.py` file suffices for training an adversarially robust CIFAR classifier with a fixed set of parameters:

```
from robustness import model_utils, datasets, train, defaults
from robustness.datasets import CIFAR
import torch as ch

# We use cox (http://github.com/MadryLab/cox) to log, store and analyze
# results. Read more at https//cox.readthedocs.io.
from cox.utils import Parameters
import cox.store

# Hard-coded dataset, architecture, batch size, workers
ds = CIFAR('/tmp/')
m, _ = model_utils.make_and_restore_model(arch='resnet50', dataset=ds)
train_loader, val_loader = ds.make_loaders(batch_size=BATCH_SIZE, workers=NUM_WORKERS)

# Create a cox store for logging
out_store = cox.store.Store(OUT_DIR)

# Hard-coded base parameters
train_kwargs = {
```

(continues on next page)

(continued from previous page)

```
    'out_dir': "train_out",
    'adv_train': 1,
    'constraint': '2',
    'eps': 0.5,
    'attack_lr': 1.5,
    'attack_steps': 20
}
train_args = Parameters(train_kwargs)

# Fill whatever parameters are missing from the defaults
train_args = defaults.check_and_fill_args(train_args,
                        defaults.TRAINING_ARGS, CIFAR)
train_args = defaults.check_and_fill_args(train_args,
                        defaults.PGD_ARGS, CIFAR)

# Train a model
train.train_model(train_args, m, (train_loader, val_loader), store=out_store)
```

The following sections will demonstrate how to customize training in a variety of ways.

2.4.1 Training networks with custom loss functions

By default, training uses the cross-entropy loss; however, we can easily change this by specifying a custom training loss and a custom adversary loss. For example, suppose that instead of just computing the cross-entropy loss, we're going to try an experimental new training loss that multiplies a random 50% of the logits by 10. (*Note that this is just for illustrative purposes—in practice this is a terrible idea.*)

We can implement this crazy loss function as a training criterion and a corresponding adversary loss. Recall that as discussed in the *robustness.train.train_model()* docstring, the train loss takes in `logits,targets` and returns a scalar, whereas the adversary loss takes in `model,inputs,targets` and returns a vector (not averaged along the batch) as well as the output.

```
train_crit = ch.nn.CrossEntropyLoss()
def custom_train_loss(logits, targ):
    probs = ch.ones_like(logits) * 0.5
    logits_to_multiply = ch.bernoulli(probs) * 9 + 1
    return train_crit(logits_to_multiply * logits, targ)

adv_crit = ch.nn.CrossEntropyLoss(reduction='none').cuda()
def custom_adv_loss(model, inp, targ):
    logits = model(inp)
    probs = ch.ones_like(logits) * 0.5
    logits_to_multiply = ch.bernoulli(probs) * 9 + 1
    new_logits = logits_to_multiply * logits
    return adv_crit(new_logits, targ), new_logits

train_args.custom_train_loss = custom_train_loss
train_args.custom_adv_loss = custom_adv_loss
```

Adding these few lines right before calling of *train_model()* suffices for training our network robustly with this custom loss.

As of the latest version of `robustness`, you can now also supply a custom function for computing accuracy using the `custom_accuracy` flag. This should be a function that takes in the model output and the target labels, and returns a tuple of `(top1, top5)` accuracies (feel free to make the second element `float('nan')` if there's only one accuracy metric you want to display). Here is an example:

```
def custom_acc_func(out, targ):
    # Calculate top1 and top5 accuracy for this batch here
    return 100., float('nan') # Return (top1, top5)

train_args.custom_accuracy = custom_acc_func
```

2.4.2 Training networks with custom data loaders

Another aspect of the training we can customize is data loading, through two utilities for modifying dataloaders called *robustness.loaders.TransformedLoader()* and *robustness.loaders.LambdaLoader*. To see how they work, we're going to consider two variations on our training: (a) training with label noise, and (b) training with random labels.

Using LambdaLoader to train with label noise

`LambdaLoader` works by modifying the output of a data loader *in real-time*, i.e. it applies a fixed function to the output of a loader. This makes it well-suited to, e.g., custom data augmentation, input/label noise, or other applications where randomness across batches is needed. To demonstrate its usage, we're going to add label noise to our training setup. To do this, all we need to do is define a function which takes in a batch of inputs and labels, and returns the same batch but with label noise added in. For example:

```
from robustness.loaders import LambdaLoader

def label_noiser(ims, labels):
    label_noise = ch.randint_like(labels, high=9)
    probs = ch.ones_like(label_noise) * 0.1
    labels_to_noise = ch.bernoulli(probs.float()).long()
    new_labels = (labels + label_noise * labels_to_noise) % 10
    return ims, new_labels

train_loader = LambdaLoader(train_loader, label_noiser)
```

Note that LamdaLoader is quite general—any function that takes in `ims, labels` and outputs `ims, labels` of the same shape can be put in place of `label_noiser` above.

Using TransformedLoader to train with random labels

In contrast to *LambdaLoader*, *TransformedLoader()* is a data loader transformation that is applied *once* at the beginning of training (this makes it better suited to deterministic transformations to inputs or labels). Unfortunately, the implementation of TransformedLoader currently loads the entire dataset into memory, so it only reliably works on small datasets (e.g. CIFAR). This will be fixed in a future version of the library. To demonstrate its usage, we will use it to randomize labels for the training set. (Recall that when we usually train using random labels, we perform the label assignment only once, prior to training.) To do this, all we need to do is define a function which takes in a batch of inputs and labels, and returns the same batch, but with random labels instead. For example:

```
from robustness.loaders import TransformedLoader
from robustness.data_augmentation import TRAIN_TRANSFORMS_DEFAULT

def make_rand_labels(ims, targs):
    new_targs = ch.randint(0, high=10, size=targs.shape).long()
    return ims, new_targs
```

(continues on next page)

(continued from previous page)

```
train_loader_transformed = TransformedLoader(train_loader,
                                             make_rand_labels,
                                             TRAIN_TRANSFORMS_DEFAULT(32),
                                             workers=NUM_WORKERS,
                                             batch_size=BATCH_SIZE,
                                             do_tqdm=True)
```

Here, we start with a `train_loader` without data augmentation, to get access to the actual image-label pairs from the training set. We then transform each input by assigning an image a random label instead. Moreover, we also support applying other transforms in *real-time* (such as data augmentation) during the creation of the transformed dataset using `train_loader_transformed` (e.g., `TRAIN_TRANSFORMS(32)` here).

Note that TransformedLoader is quite general—any function that takes in `ims, labels` and outputs `ims, labels` of the same shape can be put in place of `rand_label_transform` above.

2.4.3 Training networks with custom logging

The `robustness` library also supports training with custom logging functionality. When calling *`train_model()`*, the user can specify "hooks," functions that will be called by the training process every iteration or every epoch. Here, we'll demonstrate this functionality using a logging function that measures the norm of the network parameters (by treating them as a single vector). We will modify/augment the `main.py` code described above:

```
from torch.nn.utils import parameters_to_vector as flatten

def log_norm(mod, log_info):
   curr_params = flatten(mod.parameters())
   log_info_custom = { 'epoch': log_info['epoch'],
                       'weight_norm': ch.norm(curr_params).detach().cpu().numpy() }
   out_store['custom'].append_row(log_info_custom)
```

We now create a custom cox store that we'll hold our results in (`cox` is our super-lightweight library for storing and analyzing experimental results, you can read the docs here).

```
CUSTOM_SCHEMA = {'epoch': int, 'weight_norm': float }
out_store.add_table('custom', CUSTOM_SCHEMA)
```

We will then modify the `train_args` to incorporate this function into the logging done per epoch/iteration. If we want to log the norm of the weights every epoch, we can do:

```
train_args.epoch_hook = log_norm
```

If we want to perform the logging every iteration, we need to make the following modifications:

```
CUSTOM_SCHEMA = {'iteration': int, 'weight_norm': float}
out_store.add_table('custom', CUSTOM_SCHEMA)

 def log_norm(mod, it, loop_type, inp, targ):
    if loop_type == 'train':
       curr_params = flatten(mod.parameters())
       log_info_custom = { 'iteration': it,
                      'weight_norm': ch.norm(curr_params).detach().cpu().numpy() }
       out_store['custom'].append_row(log_info_custom)

 train_args.iteration_hook = log_norm
```

The arguments taken by the iteration hook differ from those taken by the epoch hook: the former takes a model, iteration number, loop_type, current input batch, and current target batch. The latter takes only the model and a dictionary called log_info containing all of the normally logged statistics as in train.py.

Note that the custom logging functionality provided by the robustness library is quite general—any function that takes the appropriate input arguments can be used in place of `log_norm` above.

2.4.4 Training on custom datasets

The robustness library by default includes most common datasets: ImageNet, Restricted-ImageNet, CIFAR, CINIC, and A2B. That said, it is rather straightforward to add your own dataset.

1. Subclass the *DataSet* class from *robustness.datasets*. This means implementing `__init__()` and *get_model()* functions.
2. In `__init__()`, all that is required is to call `super(NewClass, self).__init__` with the appropriate arguments, found in *the docstring* and duplicated below:

   Arguments:

   - Dataset name (e.g. `imagenet`).
   - Dataset path (if your desired dataset is in the list of already implemented datasets in torchvision.datasets, pass the appropriate location, otherwise make this an argument of your subclassed `__init__` function.

   Named arguments (all required):

   - `num_classes`, the number of classes in the dataset
   - `mean`, the mean to normalize the dataset with
   - `std`, the standard deviation to normalize the dataset with
   - `custom_class`, the *torchvision.models* class corresponding to the dataset, if it exists (otherwise `None`)
   - `label_mapping`, a dictionary mapping from class numbers to human-interpretable class names (can be `None`)
   - `transform_train`, instance of `torchvision.transforms` to apply to the training images from the dataset
   - `transform_test`, instance of `torchvision.transforms` to apply to the validation images from the dataset
3. In *get_model()*, implement a function which takes in an architecture name `arch` and boolean `pretrained`, and returns a PyTorch model (nn.Module) (see *the docstring* for more details). This will probably entail just using something like:

```
from robustness import imagenet_models # or cifar_models
assert not pretrained, "pretrained only available for ImageNet"
return imagenet_models.__dict__[arch](num_classes=self.num_classes)
# replace "models" with "cifar_models" in the above if the
# image size is less than [224, 224, 3]
```

You're all set! You can create an instance of your dataset and a corresponding model with::

```
from robustness.datasets import MyNewDataSet
from robustness.model_utils import make_and_restore_model
```

(continues on next page)

(continued from previous page)

```
ds = MyNewDataSet('/path/to/dataset/')
model, _ = make_and_restore_model(arch='resnet50', dataset=ds)
```

Note: if you also want to be able to use your dataset with the *command-line tool*, you'll need to clone the repository and pip install it locally, after also following these extra steps:

4. Add an entry to *robustness.datasets.DATASETS* dictionary for your dataset.
5. If you want to be able to train a robust model on your dataset, add it to the `DATASET_TO_CONFIG` dictionary in *main.py* and create a config file in the same manner as for the other datasets.

2.4.5 Training with custom architectures

Currently the robustness library supports a few common architectures. The models are split between two folders: `cifar_models` for architectures that handle CIFAR-size (i.e. 32x32x3) images, and `imagenet_models` for models that require larger images (e.g. 224x224x3). It is possible to add architectures to either of these folders, but to make them fully compatible with the `robustness` library requires a few extra steps.

We'll go through an example of how to add a simple one-hidden-layer MLP architecture for CIFAR:

0. Let's set up our imports and instantiate the dataset:

```
from torch import nn
from robustness.model_utils import make_and_restore_model
from robustness.datasets import CIFAR

ds = CIFAR('/path/to/cifar')
```

1. Implement and create an instance of your model:

```
class MLP(nn.Module):
    # Must implement the num_classes argument
    def __init__(self, num_classes=10):
        super().__init__()
        self.fc1 = nn.Linear(32*32*3, 1000)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(1000, num_classes)

    def forward(self, x, *args, **kwargs):
        out = x.view(x.shape[0], -1)
        out = self.fc1(out)
        out = self.relu1(out)
        return self.fc2(out)

model = MLP(num_classes=10)
```

2. Call *robustness.model_utils.make_and_restore_model()*, but this time feed in `model` instead of a string with the architecture name:

```
model, _ = make_and_restore_model(arch=model, dataset=ds)
```

3. (If all you want to do with this architecture is training a model, **you can skip this step**). In order to make it fully compatible with the robustness library, the *:forward* function of our architecture must support the following three (boolean) arguments:

    - `with_latent` : If this option is given, `forward` should return the output of the second-last layer along with the logits.

- `fake_relu` : If this option is given, replace the ReLU just after the second-last layer with a `custom_modules.FakeReLUM`, which is a ReLU on the forwards pass and identity on the backwards pass.
- `no_relu` : If this option is given, then `with_latent` should return the *pre-ReLU* activations of the second-last layer.

These options are usually actually quite simple to implement:

```
from robustness.imagenet_models import custom_modules

class MLP(nn.Module):
   # Must implement the num_classes argument
   def __init__(self, num_classes=10):
      super().__init__()
      self.fc1 = nn.Linear(32*32*3, 1000)
      self.relu1 = nn.ReLU()
      self.fake_relu1 = custom_modules.FakeReLUM()
      self.fc2 = nn.Linear(1000, num_classes)

   def forward(self, x, with_latent=False, fake_relu=False, no_relu=False):
      out = x.view(x.shape[0], -1)
      pre_relu = self.fc1(out)
      out = self.fake_relu1(pre_relu) if fake_relu else self.relu1(pre_relu)
      final = self.fc2(out)
      if with_latent:
         return (final, pre_relu) if no_relu else (final, out)
      return final
```

That's it! Now, just like for custom datasets, if you want these architectures to be available via the *command line tool*, you'll have to clone the `robustness` repository and pip install it locally. You'll also have to do the following:

4. Put the declaration of the `MLP` class into its own `mlp.py` file, and add this file to the `cifar_models` folder
3. In `cifar_models/__init__.py`, add the line:

```
from .mlp import MLP
```

4. The new architecture is now available as:

```
from robustness.model_utils import make_and_restore_model
from robustness.datasets import CIFAR
ds = CIFAR('/path/to/cifar')
model, _ = make_and_restore_model(arch='MLP', dataset=ds)
```

and via the command line option `--arch MLP`.

2.5 Creating a custom dataset by superclassing ImageNet

Often in both adversarial robustness research and otherwise, datasets with the richness of ImageNet are desired, but without the added complexity of the 1000-way ILSVRC classification task. A common workaround is to "superclass" ImageNet, that is, to define a new dataset that contains broad classes which each subsume several of the original ImageNet classes.

In this document, we will discuss how to (a) load pre-packaged ImageNet-based datasets that we've created, and (b) create new custom N-class subset of ImageNet data by leveraging the WordNet hierarchy to build superclasses. The robustness library provides functionality to do this via the *`CustomImageNet`* and `ImageNetHierarchy` classes.

In this walkthrough, we'll see how to use these classes to browse and use the WordNet hierarchy to create custom ImageNet-based datasets.

2.5.1 Requirements/Setup

To create custom ImageNet datasets, we need (a) the ImageNet dataset to be downloaded and available in PyTorch-readable format, and (b) the files `wordnet.is_a.txt`, `words.txt` and `imagenet_class_index.json`, all contained within the same directory (all of these files can be obtained from the ImageNet website.

2.5.2 Basic Usage: Loading Pre-Packaged ImageNet-based Datasets

To make things as easy as possible, we've compiled a list of large, but less complex ImageNet-based datasets. These datasets can be loaded in their unbalanced or balanced forms, where in the latter we truncate each class to have the same number of images as the smallest class. We enumerate these datasets below:

Dataset Name	Classes
living_9	Dog (n02084071), Bird (n01503061), Arthropod (n01767661), Reptile (n01661091), Primate (n02469914), Fish (n02512053), Feline (n02120997), Bovid (n02401031), Amphibian (n01627424)
mixed_10	Dog (n02084071), Bird (n01503061), Insect (n02159955), Monkey (n02484322), Car (n02958343), Cat (n02120997), Truck (n04490091), Fruit (n13134947), Fungus (n12992868), Boat (n02858304)
mixed_13	Dog (n02084071), Bird (n01503061), Insect (n02159955), Furniture (n03405725), Fish (n02512053), Monkey (n02484322), Car (n02958343), Cat (n02120997), Truck (n04490091), Fruit (n13134947), Fungus (n12992868), Boat (n02858304), Computer (n03082979)
geirhos_16	Aircraft (n02686568), Bear (n02131653), Bicycle (n02834778), Bird (n01503061), Boat (n02858304), Bottle (n02876657), Car (n02958343), Cat (n02121808), Char (n03001627), Clock (n03046257), Dog (n02084071), Elephant (n02503517), Keyboard (n03614532), Knife (n03623556), Oven (n03862676), Truck (n04490091),
big_12	Dog (n02084071), Structure(n04341686), Bird (n01503061), Clothing (n03051540), Vehicle(n04576211), Reptile (n01661091), Carnivore (n02075296), Insect (n02159955), Instrument (n03800933), Food (n07555863), Furniture (n03405725), Primate (n02469914),

Loading any of these datasets (for example, `mixed_10`) is relatively simple:

```
from robustness import datasets
from robustness.tools.imagenet_helpers import common_superclass_wnid,␣
↪ImageNetHierarchy

in_hier = ImageNetHierarchy(in_path, in_info_path)
superclass_wnid = common_superclass_wnid('mixed_10')
class_ranges, label_map = in_hier.get_subclasses(superclass_wnid, balanced=True)
```

In the above, `in_path` should point to a folder with the ImageNet dataset in `train` and `val` sub-folders; `in_info_path` should be the path to the directory containing the aforementioned files (`wordnet.is_a.txt`, `words.txt`, `imagenet_class_index.json`).

We can then create a dataset and the corresponding data loader using:

```
custom_dataset = datasets.CustomImageNet(in_path, class_ranges)
train_loader, test_loader = custom_dataset.make_loaders(workers=num_workers,
                                                        batch_size=batch_size)
```

You're all set! You can then use this `custom_dataset` and loaders just as you would any other existing/custom dataset in the robustness library. For instance, you can visualize training set samples and their labels using:

```
from robustness.tools.vis_tools import show_image_row
im, lab = next(iter(train_loader))
show_image_row([im], tlist=[[label_map[int(k)] for k in lab]])
```

2.5.3 Advanced Usage (Making Custom Datasets) Part 1: Browsing the WordNet Hierarchy

The `ImageNetHierarchy` class allows us to probe the WordNet hierarchy and create custom datasets with the desired number of superclasses. We first create an instance of the `ImageNetHierarchy` class:

```
from robustness.tools.imagenet_helpers import ImageNetHierarchy
in_hier = ImageNetHierarchy(in_path, in_info_path)
```

Again, `in_path` should point to a folder with the ImageNet dataset in `train` and `val` sub-folders; `in_info_path` should be the path to the directory containing the aforementioned files (`wordnet.is_a.txt`, `words.txt`, `imagenet_class_index.json`).

We can now use the `in_hier` object to probe the ImageNet hierarchy. The `wnid_sorted` attribute, for example, is an iterator over the WordNet IDs, sorted by the number of descendents they have which are ImageNet classes:

```
for cnt, (wnid, ndesc_in, ndesc_total) in enumerate(in_hier.wnid_sorted):
    print(f"WordNet ID: {wnid}, Name: {in_hier.wnid_to_name[wnid]}, #ImageNet␣
↪descendants: {ndesc_in}")
```

Given any WordNet ID, we can also enumerate all of its subclasses of a given superclass using the `in_hier.tree` object and its related methods/attributes:

```
ancestor_wnid = 'n02120997'
print(f"Superclass | WordNet ID: {ancestor_wnid}, Name: {in_hier.wnid_to_
↪name[ancestor_wnid]}")

for cnt, wnid in enumerate(in_hier.tree['n02120997'].descendants_all):
    print(f"Subclass | WordNet ID: {wnid}, Name: {in_hier.wnid_to_name[wnid]}")
```

We can filter these subclasses based on whether they correspond to ImageNet classes using the `in_wnids` attribute:

```
ancestor_wnid = 'n02120997'
print(f"Superclass | WordNet ID: {ancestor_wnid}, Name: {in_hier.wnid_to_
↪name[ancestor_wnid]}")
for cnt, wnid in enumerate(in_hier.tree[ancestor_wnid].descendants_all):
    if wnid in in_hier.in_wnids:
        print(f"ImageNet subclass | WordNet ID: {wnid}, Name: {in_hier.wnid_to_
↪name[wnid]}")
```

2.5.4 Advanced Usage (Making Custom Datasets) Part 2: Making the Datasets

To create a dataset with the desired number of superclasses we use the `get_superclasses()` function, which takes in a desired number of superclasses `n_classes`, an (optional) WordNet ID `ancestor_wnid` that allows us to fix a common WordNet ancestor for all the classes in our new dataset, and an optional boolean `balanced` to get a balanced dataset (where each superclass has the same number of ImageNet subclasses). (see `the docstring` for more details).

```
superclass_wnid, class_ranges, label_map = in_hier.get_superclasses(n_classes,
                                            ancestor_wnid=ancestor_wnid,
                                            balanced=balanced)
```

This method returns WordNet IDs of chosen superclasses `superclass_wnid`, sets of ImageNet subclasses to group together for each of the superclasses `class_ranges`, and a mapping from superclass number to its human-interpretable description `label_map`.

You can also directly provide a list of superclass WordNet IDs `ancestor_wnid` that you would like to use to build a custom dataset. For instance, some sample superclass groupings can be found in py:meth:*~robustness.tools.imagenet_helpers.ImageNetHierarchy.common_superclass_wnid*.

Once a list of WordNet IDs has been acquired (whether through the method described here or just manually), we can use the method presented at the beginning of this article to load the corresponding dataset:

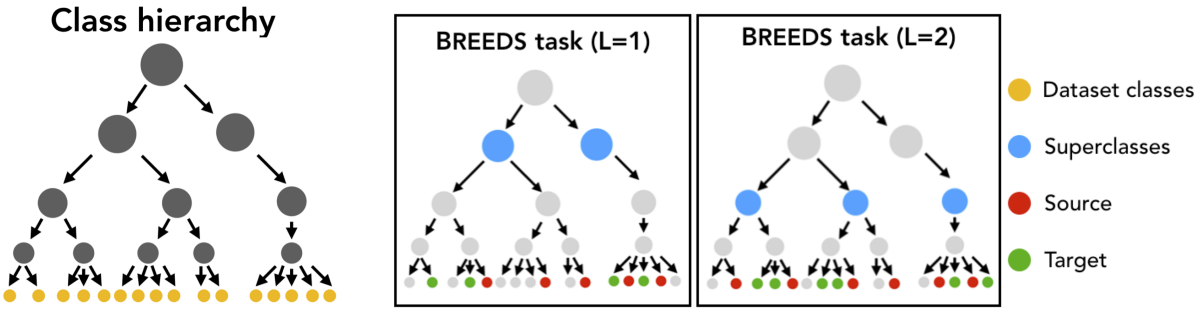
```
custom_dataset = datasets.CustomImageNet(in_path, class_ranges)
train_loader, test_loader = custom_dataset.make_loaders(workers=num_workers,
                                                        batch_size=batch_size)
```

2.6 Creating BREEDS subpopulation shift benchmarks

In this document, we will discuss how to create BREEDS datasets [STM20]. Given any existing dataset that comes with a class hierarchy (e.g. ImageNet, OpenImages), the BREEDS methodology allows you to make a derivative classification task that can be used to measure robustness to subpopulation shift. To do this, we:

1. Group together semantically-simlar classes ("breeds") in the dataset into superclasses.
2. Define a classification task in terms of these superclasses—with the twist that the "breeds" used in the training set from each superclasses are disjoint from the "breeds" used in the test set.

As a primitive example, one could take ImageNet (which contains many classes corresponding to cat and dog breeds), and use the BREEDS methodology to come up with a derivative "cats vs. dogs" task, where the training set would contain one set of breeds (e.g., Egyptian cat and Tabby Cat vs. Labrador and Golden Retriever) and the test set would contain another set (e.g. Persian cat and alley cat vs Mastiff and Poodle). Here is a pictorial illustration of the BREEDS approach:



This methodology allows you to create subpopulation shift benchmarks of varying difficulty automatically, without having to manually group or split up classes, and can be applied to any dataset which has a class hierarchy. In this walkthrough, we will use ImageNet and the corresponding class hierarchy from [STM20].

2.6.1 Requirements/Setup

To create BREEDS datasets using ImageNet, we need to create a:

- `data_dir` which contains the ImageNet dataset in PyTorch-readable format.
- `info_dir` which contains the following information (files) about the class hierarchy:
  - `dataset_class_info.json`: A list whose entries are triplets of class number, class ID and class name, for each dataset class.
  - `class_hierarchy.txt`: Every line denotes an edge—parent ID followed by child ID (space separated)—in the class hierarchy.
  - `node_names.txt`: Each line contains the ID of a node followed by it's name (tab separated).

For convenience, we provide the relevant files for the (modified) class hierarchy here. You can manually download them and move them to `info_dir` or do it automatically by specifying an empty `info_dir` to `get_superclasses()`:

```
from robustness.tools.breeds_helpers import setup_breeds

setup_breeds(info_dir)
```

2.6.2 Part 1: Browsing through the Class Hierarchy

We can use `ClassHierarchy` to examine a dataset's (here, ImageNet) class hierarchy. Here, `info_dir` should contain the requisite files for the class hierarchy (from the Setup step):

```
from robustness.tools.breeds_helpers import ClassHierarchy
import numpy as np

hier = ClassHierarchy(info_dir)
print(f"# Levels in hierarchy: {np.max(list(hier.level_to_nodes.keys()))}")
print(f"# Nodes/level:",
    [f"Level {k}: {len(v)}" for k, v in hier.level_to_nodes.items()])
```

The `hier` object has a `graph` attribute, which represents the class hierarchy as a `networkx` graph. In this graph, the children of a node correspond to its subclasses (e.g., Labrador would be a child of the dog class in our primitive example). Note that all the original dataset classes will be the leaves of this graph.

We can then use this graph to define superclasses—all nodes at a user-specified depth from the root node. For example:

```
level = 2 # Could be any number smaller than max level
superclasses = hier.get_nodes_at_level(level)
print(f"Superclasses at level {level}:\n")
print(", ".join([f"{hier.HIER_NODE_NAME[s]}" for s in superclasses]))
```

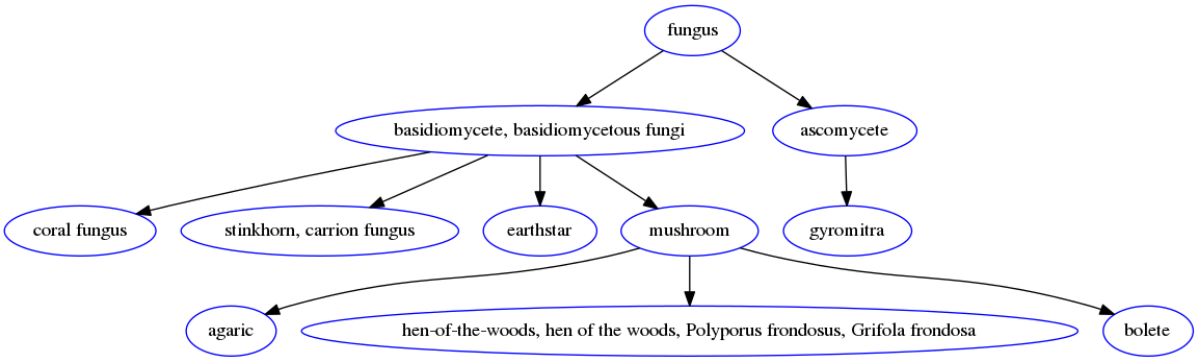
Each superclass is made up of multiple "breeds", which simply correspond to the leaves (original dataset classes) that are its descendants in the class hierarchy:

```
idx = np.random.randint(0, len(superclasses), 1)[0]
superclass = list(superclasses)[idx]
subclasses = hier.leaves_reachable(superclass)
print(f"Superclass: {hier.HIER_NODE_NAME[superclass]}\n")

print(f"Subclasses ({len(subclasses)}):")
print([f"{hier.LEAF_ID_TO_NAME[l]}" for l in list(subclasses)])
```

We can also visualize subtrees of the graph with the help of the *networkx* and *pygraphviz* packages. For instance, we can taks a look at the subtree of the class hierarchy rooted at a particular superclass:

```
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout, to_agraph
import pygraphviz as pgv
from IPython.display import Image

subtree = nx.ego_graph(hier.graph, superclass, radius=10)
mapping = {n: hier.HIER_NODE_NAME[n] for n in subtree.nodes()}
subtree = to_agraph(nx.relabel_nodes(subtree, mapping))
subtree.delete_edge(subtree.edges()[0])
subtree.layout('dot')
subtree.node_attr['color']='blue'
subtree.draw('graph.png', format='png')
Image('graph.png')
```

For instance, visualizing tree rooted at the `fungus` superclass yields:



2.6.3 Part 2: Creating BREEDS Datasets

To create a dataset composed of superclasses, we use the `BreedsDatasetGenerator`. Internally, this class instantiates an object of `ClassHierarchy` and uses it to define the superclasses.

```
from robustness.tools.breeds_helpers import BreedsDatasetGenerator

DG = BreedsDatasetGenerator(info_dir)
```

Specifically, we will use `get_superclasses()`. This function takes in the following arguments (see `this docstring` for more details):

- `level`: Level in the hierarchy (in terms of distance from the root node) at which to define superclasses.
- `Nsubclasses`: Controls the minimum number of subclasses/superclass in the dataset. If None, it is automatically set to be the size (in terms of subclasses) of the smallest superclass.
- `split`: If `None`, subclasses of a superclass are returned as is, without partitioning them into the source and target domains. Else, can be `rand/good/bad` depending on whether the subclass split should be random or less/more adversarially chosen (see paper for details).
- `ancestor`: If a node ID is specified, superclasses are chosen from subtree of class hierarchy rooted at this node. Else, if None, `ancestor` is set to be the root node.
- `balanced`: If True, subclasses/superclass is fixed over superclasses.

For instance, we could create a balanced dataset, with the subclass partition being less adversarial as follows:

```
ret = DG.get_superclasses(level=2,
                          Nsubclasses=None,
                          split="rand",
                          ancestor=None,
                          balanced=True)
superclasses, subclass_split, label_map = ret
```

This method returns:

- `superclasses` is a list containing the IDs of all the superclasses.
- `subclass_split` is a tuple of subclass ranges for the source and target domains. For instance, `subclass_split[0]` is a list, which for each superclass, contains a list of subclasses present in the source domain. If `split=None`, subclass_split[1] is empty and can be ignored.
- `label_map` is a dictionary mapping a superclass number (label) to name.

You can experiment with these parameters to create datasets of different granularity. For instance, you could specify the `Nsubclasses` to restrict the size of every superclass in the dataset, set the `ancestor` to be a specific node (e.g., `n00004258` to focus on living things), or set `balanced` to `False` to get an imbalanced dataset.

We can take a closer look at the composition of the dataset—what superclasses/subclasses it contains—using:

```
from robustness.tools.breeds_helpers import print_dataset_info

print_dataset_info(superclasses,
                   subclass_split,
                   label_map,
                   hier.LEAF_NUM_TO_NAME)
```

Finally, for the source and target domains, we can create datasets and their corresponding loaders:

```
from robustness import datasets

train_subclasses, test_subclasses = subclass_split

dataset_source = datasets.CustomImageNet(data_dir, train_subclasses)
loaders_source = dataset_source.make_loaders(num_workers, batch_size)
train_loader_source, val_loader_source = loaders_source

dataset_target = datasets.CustomImageNet(data_dir, test_subclasses)
loaders_target = dataset_source.make_loaders(num_workers, batch_size)
train_loader_target, val_loader_target = loaders_target
```

You're all set! You can then use this dataset and loaders just as you would any other existing/custom dataset in the robustness library. For instance, you can visualize validation set samples from both domains and their labels using:

```
from robustness.tools.vis_tools import show_image_row

for domain, loader in zip(["Source", "Target"],
                          [val_loader_source, val_loader_target]):
    im, lab = next(iter(loader))
    show_image_row([im],
                   tlist=[[label_map[int(k)].split(",")[0] for k in lab]],
                   ylist=[domain],
                   fontsize=20)
```

You can also create superclass tasks where subclasses are not partitioned across domains:

```
ret = DG.get_superclasses(level=2,
                          Nsubclasses=2,
                          split=None,
                          ancestor=None,
                          balanced=True)
superclasses, subclass_split, label_map = ret
all_subclasses = subclass_split[0]

dataset = datasets.CustomImageNet(data_dir, all_subclasses)

print_dataset_info(superclasses,
                   subclass_split,
                   label_map,
                   hier.LEAF_NUM_TO_NAME)
```

2.6.4 Part 3: Loading in-built BREEDS Datasets

Alternatively, we can directly use one of the datasets from our paper [STM20]—namely `Entity13`, `Entity30`, `Living17` and `Nonliving26`. Loading any of these datasets is relatively simple:

```
from robustness.tools.breeds_helpers import make_living17
ret = make_living17(info_dir, split="rand")
superclasses, subclass_split, label_map = ret

print_dataset_info(superclasses,
                   subclass_split,
                   label_map,
                   hier.LEAF_NUM_TO_NAME)
```

You can then use a similar methodology to Part 2 above to probe dataset information and create datasets and loaders.

2.7 CHANGELOG

2.7.1 robustness 1.2.post2

- Add SqueezeNet architectures
- (Preliminary) Torch 1.7 support
- Support for specifying device_ids in DataParallel

2.7.2 robustness 1.2

- **Biggest new features:**
  - New ImageNet models
  - Mixed-precision training
  - OpenImages and Places365 datasets added
  - **Ability to specify a custom accuracy function (custom loss functions** were already supported, this is just for logging)
  - Improved resuming functionality
- **Changes to CLI-based training:**
  - `--custom-lr-schedule` replaced by `--custom-lr-multiplier` (same format)
  - **`--eps-fadein-epochs` replaced by general `--custom-eps-multiplier`** (now same format as custom-lr schedule)
  - **`--step-lr-gamma` now available to change the size of learning rate** drops (used to be fixed to 10x drops)
  - **`--lr-interpolation` argument added (can choose between linear and step** interpolation between learning rates in the schedule)
  - **`--weight_decay` is now called `--weight-decay`, keeping with** convention
  - **`--resume-optimizer` is a 0/1 argument for whether to resume the** optimizer and LR schedule, or just the model itself
  - **`--mixed-precision` is a 0/1 argument for whether to use mixed-precision** training or not (required PyTorch compiled with AMP support)
- **Model and data loading:**
  - **DataParallel is now *off* by default when loading models, even when** resume_path is specified (previously it was off for new models, and on for resumed models by default)
  - **New `add_custom_forward` for `make_and_restore_model` (see docs for** more details)
  - Can now pass a random seed for training data subsetting
- **Training:**
  - See new CLI features—most have training-as-a-library counterparts
  - Fixed a bug that did not resume the optimizer and schedule
  - Support for custom accuracy functions
  - **Can now disable `torch.nograd` for test set eval (in case you have a** custom accuracy function that needs gradients even on the val set)
- **PGD:**
  - Better random start for l2 attacks
  - **Added a `RandomStep` attacker step (useful for large-noise training with** varying noise over training)
  - Fixed bug in the `with_image` argument (minor)
- **Model saving:**
  - **Accuracies are now saved in the checkpoint files themselves (instead of** just in the log stores)

  - **Removed redundant checkpoints table from the log store, as it is a** duplicate of the latest checkpoint file and just wastes space

- **Cleanup:**
  - Remove redundant `save_checkpoint` function in helpers file
  - Code flow improvements

2.7.3 robustness 1.1.post2

- Critical fix in *`robustness.loaders.TransformedLoader()`*, allow for data shuffling

2.7.4 robustness 1.1

- Added ability to superclass ImageNet to make custom datasets (*docs*)
- Added `shuffle_train` and `shuffle_test` options to `make_loaders()`
- Added support for cyclic learning rate (`--custom-schedule cyclic` via command line or `{"custom_schedule": "cyclic"}` from Python
- Added support for transfer learning/partial parameter updates, *`robustness.train.train_model()`* now takes `update_params` argument, list of parameters to update
- Allow `random_start` (random start for adversarial attacks) to be set via command line
- Change defaults for ImageNet training (`200` epochs instead of `350`)
- Small fixes/refinements to *`robustness.tools.vis_tools`* module

CHAPTER 3

API Reference

We provide an API reference where we discuss the role of each module and provide extensive documentation.

3.1 API Reference

3.1.1 robustness.attack_steps module

For most use cases, this can just be considered an internal class and ignored.

This module contains the abstract class AttackerStep as well as a few subclasses.

AttackerStep is a generic way to implement optimizers specifically for use with *robustness.attacker.AttackerModel*. In general, except for when you want to *create a custom optimization method*, you probably do not need to import or edit this module and can just think of it as internal.

class robustness.attack_steps.**AttackerStep**(*orig_input*, *eps*, *step_size*, *use_grad=True*)

Bases: object

Generic class for attacker steps, under perturbation constraints specified by an "origin input" and a perturbation magnitude. Must implement project, step, and random_perturb

Initialize the attacker step with a given perturbation magnitude.

Parameters

- **eps** (*float*) – the perturbation magnitude
- **orig_input** (*ch.tensor*) – the original input

project(*x*)

Given an input x, project it back into the feasible set

Parameters **x** (*ch.tensor*) – the input to project back into the feasible set.

Returns A *ch.tensor* that is the input projected back into the feasible set, that is,

$$\min_{x' \in S} \|x' - x\|_2$$

step (*x*, *g*)
Given a gradient, make the appropriate step according to the perturbation constraint (e.g. dual norm maximization for ℓ_p norms).

Parameters **g** (*ch.tensor*) – the raw gradient

Returns The new input, a ch.tensor for the next step.

random_perturb (*x*)
Given a starting input, take a random step within the feasible set

to_image (*x*)
Given an input (which may be in an alternative parameterization), convert it to a valid image (this is implemented as the identity function by default as most of the time we use the pixel parameterization, but for alternative parameterizations this functino must be overriden).

class robustness.attack_steps.**LinfStep**(*orig_input*, *eps*, *step_size*, *use_grad=True*)
Bases: *robustness.attack_steps.AttackerStep*

Attack step for ℓ_∞ threat model. Given x_0 and ϵ, the constraint set is given by:

$$S = \{x | \|x - x_0\|_\infty \leq \epsilon\}$$

Initialize the attacker step with a given perturbation magnitude.

Parameters

- **eps** (*float*) – the perturbation magnitude
- **orig_input** (*ch.tensor*) – the original input

project (*x*)

step (*x*, *g*)

random_perturb (*x*)

class robustness.attack_steps.**L2Step**(*orig_input*, *eps*, *step_size*, *use_grad=True*)
Bases: *robustness.attack_steps.AttackerStep*

Attack step for ℓ_∞ threat model. Given x_0 and ϵ, the constraint set is given by:

$$S = \{x | \|x - x_0\|_2 \leq \epsilon\}$$

Initialize the attacker step with a given perturbation magnitude.

Parameters

- **eps** (*float*) – the perturbation magnitude
- **orig_input** (*ch.tensor*) – the original input

project (*x*)

step (*x*, *g*)

random_perturb (*x*)

class robustness.attack_steps.**UnconstrainedStep**(*orig_input*, *eps*, *step_size*, *use_grad=True*)
Bases: *robustness.attack_steps.AttackerStep*

Unconstrained threat model, $S = [0, 1]^n$.

Initialize the attacker step with a given perturbation magnitude.

Parameters

- **eps** (*float*) – the perturbation magnitude
- **orig_input** (*ch.tensor*) – the original input

project(*x*)

step(*x*, *g*)

random_perturb(*x*)

class robustness.attack_steps.**FourierStep**(*orig_input*, *eps*, *step_size*, *use_grad=True*)
Bases: *robustness.attack_steps.AttackerStep*

Step under the Fourier (decorrelated) parameterization of an image.

See https://distill.pub/2017/feature-visualization/#preconditioning for more information.

Initialize the attacker step with a given perturbation magnitude.

Parameters

- **eps** (*float*) – the perturbation magnitude
- **orig_input** (*ch.tensor*) – the original input

project(*x*)

step(*x*, *g*)

random_perturb(*x*)

to_image(*x*)

class robustness.attack_steps.**RandomStep**(**args*, ***kwargs*)
Bases: *robustness.attack_steps.AttackerStep*

Step for Randomized Smoothing.

project(*x*)

step(*x*, *g*)

random_perturb(*x*)

3.1.2 robustness.attacker module

For most use cases, this can just be considered an internal class and ignored.

This module houses the *robustness.attacker.Attacker* and *robustness.attacker.AttackerModel* classes.

Attacker is an internal class that should not be imported/called from outside the library. *AttackerModel* is a "wrapper" class which is fed a model and adds to it adversarial attack functionalities as well as other useful options. See *robustness.attacker.AttackerModel.forward()* for documentation on which arguments AttackerModel supports, and see *robustness.attacker.Attacker.forward()* for the arguments pertaining to adversarial examples specifically.

For a demonstration of this module in action, see the walkthrough "*Input manipulation with pre-trained models*"

Note 1: `.forward()` should never be called directly but instead the AttackerModel object itself should be called, just like with any `nn.Module` subclass.

Note 2: Even though the adversarial example arguments are documented in *robustness.attacker.Attacker.forward()*, this function should never be called directly—instead, these arguments are passed along from *robustness.attacker.AttackerModel.forward()*.

class robustness.attacker.**Attacker**(*model*, *dataset*)

Bases: sphinx.ext.autodoc.importer._MockObject

Attacker class, used to make adversarial examples.

This is primarily an internal class, you probably want to be looking at *robustness.attacker.AttackerModel*, which is how models are actually served (AttackerModel uses this Attacker class).

However, the robustness.Attacker.forward() function below documents the arguments supported for adversarial attacks specifically.

Initialize the Attacker

Parameters

- **model** (*nn.Module*) – the PyTorch model to attack
- **dataset** (*Dataset*) – dataset the model is trained on, only used to get mean and std for normalization

forward(*x*, *target*, **_*, *constraint*, *eps*, *step_size*, *iterations*, *random_start=False*, *random_restarts=False*, *do_tqdm=False*, *targeted=False*, *custom_loss=None*, *should_normalize=True*, *orig_input=None*, *use_best=True*, *return_image=True*, *est_grad=None*, *mixed_precision=False*)

Implementation of forward (finds adversarial examples). Note that this does **not** perform inference and should not be called directly; refer to *robustness.attacker.AttackerModel.forward()* for the function you should actually be calling.

Parameters

- **target** (*x,*) – see *robustness.attacker.AttackerModel.forward()*
- **constraint** – ("2"|"inf"|"unconstrained"|"fourier"|*AttackerStep*) : threat model for adversarial attacks (ℓ_2 ball, ℓ_∞ ball, $[0, 1]^n$, Fourier basis, or custom AttackerStep subclass).
- **eps** (*float*) – radius for threat model.
- **step_size** (*float*) – step size for adversarial attacks.
- **iterations** (*int*) – number of steps for adversarial attacks.
- **random_start** (*bool*) – if True, start the attack with a random step.
- **random_restarts** (*bool*) – if True, do many random restarts and take the worst attack (in terms of loss) per input.
- **do_tqdm** (*bool*) – if True, show a tqdm progress bar for the attack.
- **targeted** (*bool*) – if True (False), minimize (maximize) the loss.
- **custom_loss** (*function|None*) – if provided, used instead of the criterion as the loss to maximize/minimize during adversarial attack. The function should take in model, x, target and return a tuple of the form loss, None, where loss is a tensor of size N (per-element loss).
- **should_normalize** (*bool*) – If False, don't normalize the input (not recommended unless normalization is done in the custom_loss instead).
- **orig_input** (*ch.tensor|None*) – If not None, use this as the center of the perturbation set, rather than x.
- **use_best** (*bool*) – If True, use the best (in terms of loss) iterate of the attack process instead of just the last one.

- **return_image** (*bool*) – If True (default), then return the adversarial example as an image, otherwise return it in its parameterization (for example, the Fourier coefficients if 'constraint' is 'fourier')
- **est_grad** (*tuple/None*) – If not None (default), then these are `(query_radius [R], num_queries [N])` to use for estimating the gradient instead of autograd. We use the spherical gradient estimator, shown below, along with antithetic sampling[1] to reduce variance: $\nabla_x f(x) \approx \sum_{i=0}^{N} f(x + R \cdot \vec{\delta_i}) \cdot \vec{\delta_i}$, where δ_i are randomly sampled from the unit ball.
- **mixed_precision** (*bool*) – if True, use mixed-precision calculations to compute the adversarial examples / do the inference.

Returns

An adversarial example for x (i.e. within a feasible set determined by *eps* and *constraint*, but classified as:

- *target* (if *targeted* == *True*)
- not *target* (if *targeted* == *False*)

class robustness.attacker.**AttackerModel**(*model*, *dataset*)

Bases: sphinx.ext.autodoc.importer._MockObject

Wrapper class for adversarial attacks on models. Given any normal model (a `ch.nn.Module` instance), wrapping it in AttackerModel allows for convenient access to adversarial attacks and other applications.:

```
model = ResNet50()
model = AttackerModel(model)
x = ch.rand(10, 3, 32, 32) # random images
y = ch.zeros(10) # label 0
out, new_im = model(x, y, make_adv=True) # adversarial attack
out, new_im = model(x, y, make_adv=True, targeted=True) # targeted attack
out = model(x) # normal inference (no label needed)
```

More code examples available in the documentation for *forward*. For a more comprehensive overview of this class, see *our detailed walkthrough*.

forward(*inp*, *target=None*, *make_adv=False*, *with_latent=False*, *fake_relu=False*, *no_relu=False*, *with_image=True*, ***attacker_kwargs*)

Main function for running inference and generating adversarial examples for a model.

Parameters

- **inp** (*ch.tensor*) – input to do inference on [N x input_shape] (e.g. NCHW)
- **target** (*ch.tensor*) – ignored if *make_adv* == *False*. Otherwise, labels for adversarial attack.
- **make_adv** (*bool*) – whether to make an adversarial example for the model. If true, returns a tuple of the form `(model_prediction, adv_input)` where `model_prediction` is a tensor with the *logits* from the network.
- **with_latent** (*bool*) – also return the second-last layer along with the logits. Output becomes of the form `((model_logits, model_layer), adv_input)` if `make_adv==True`, otherwise `(model_logits, model_layer)`.
- **fake_relu** (*bool*) – useful for activation maximization. If `True`, replace the ReLUs in the last layer with "fake ReLUs," which are ReLUs in the forwards pass but identity in

[1] This means that we actually draw $N/2$ random vectors from the unit ball, and then use $\delta_{N/2+i} = -\delta_i$.

the backwards pass (otherwise, maximizing a ReLU which is dead is impossible as there is no gradient).

- **no_relu** (*bool*) – If `True`, return the latent output with the (pre-ReLU) output of the second-last layer, instead of the post-ReLU output. Requires `fake_relu=False`, and has no visible effect without `with_latent=True`.
- **with_image** (*bool*) – if `False`, only return the model output (even if `make_adv == True`).

3.1.3 robustness.data_augmentation module

Module responsible for data augmentation constants and configuration.

class robustness.data_augmentation.**Lighting**(*alphastd*, *eigval*, *eigvec*)
Bases: object

Lighting noise (see https://git.io/fhBOc)

robustness.data_augmentation.**TRAIN_TRANSFORMS_IMAGENET**
Random crop, Random flip, Color Jitter, and Lighting Transform (see https://git.io/fhBOc)

Type Standard training data augmentation for ImageNet-scale datasets

robustness.data_augmentation.**TEST_TRANSFORMS_IMAGENET**
Standard test data processing (no augmentation) for ImageNet-scale datasets, Resized to 256x256 then center cropped to 224x224.

robustness.data_augmentation.**TRAIN_TRANSFORMS_DEFAULT**(*size*)
Generic training data transform, given image side length does random cropping, flipping, color jitter, and rotation. Called as, for example, `robustness.data_augmentation.TRAIN_TRANSFORMS_DEFAULT(32)()` for CIFAR-10.

robustness.data_augmentation.**TEST_TRANSFORMS_DEFAULT**(*size*)
Generic test data transform (no augmentation) to complement *robustness.data_augmentation.TEST_TRANSFORMS_DEFAULT()*, takes in an image side length.

3.1.4 robustness.datasets module

Module containing all the supported datasets, which are subclasses of the abstract class *robustness.datasets.DataSet*.

Currently supported datasets:

- ImageNet (*robustness.datasets.ImageNet*)
- RestrictedImageNet (*robustness.datasets.RestrictedImageNet*)
- CIFAR-10 (*robustness.datasets.CIFAR*)
- CINIC-10 (*robustness.datasets.CINIC*)
- A2B: horse2zebra, summer2winter_yosemite, apple2orange (*robustness.datasets.A2B*)

Using robustness as a general training library (Part 2: Customizing training) shows how to add custom datasets to the library.

class robustness.datasets.**DataSet**(*ds_name*, *data_path*, ***kwargs*)
Bases: object

Base class for representing a dataset. Meant to be subclassed, with subclasses implementing the *get_model* function.

Parameters

- **ds_name** (*str*) – string identifier for the dataset
- **data_path** (*str*) – path to the dataset
- **num_classes** (*int*) – *required kwarg*, the number of classes in the dataset
- **mean** (*ch.tensor*) – *required kwarg*, the mean to normalize the dataset with (e.g. `ch.tensor([0.4914, 0.4822, 0.4465])` for CIFAR-10)
- **std** (*ch.tensor*) – *required kwarg*, the standard deviation to normalize the dataset with (e.g. `ch.tensor([0.2023, 0.1994, 0.2010])` for CIFAR-10)
- **custom_class** (*type*) – *required kwarg*, a `torchvision.models` class corresponding to the dataset, if it exists (otherwise `None`)
- **label_mapping** (*dict[int,str]*) – *required kwarg*, a dictionary mapping from class numbers to human-interpretable class names (can be `None`)
- **transform_train** (*torchvision.transforms*) – *required kwarg*, transforms to apply to the training images from the dataset
- **transform_test** (*torchvision.transforms*) – *required kwarg*, transforms to apply to the validation images from the dataset

override_args(*default_args*, *kwargs*)
Convenience method for overriding arguments. (Internal)

get_model(*arch*, *pretrained*)
Should be overriden by subclasses. Also, you will probably never need to call this function, and should instead by using model_utils.make_and_restore_model.

Parameters

- **arch** (*str*) – name of architecture
- **pretrained** (*bool*) – whether to try to load torchvision pretrained checkpoint

Returns A model with the given architecture that works for each dataset (e.g. with the right input/output dimensions).

make_loaders(*workers*, *batch_size*, *data_aug=True*, *subset=None*, *subset_start=0*, *subset_type='rand'*, *val_batch_size=None*, *only_val=False*, *shuffle_train=True*, *shuffle_val=True*, *subset_seed=None*)

Parameters

- **workers** (*int*) – number of workers for data fetching (*required*). batch_size (int) : batch size for the data loaders (*required*).
- **data_aug** (*bool*) – whether or not to do train data augmentation.
- **subset** (*None|int*) – if given, the returned training data loader will only use a subset of the training data; this should be a number specifying the number of training data points to use.
- **subset_start** (*int*) – only used if *subset* is not None; this specifies the starting index of the subset.
- **subset_type** (*"rand"|"first"|"last"*) – only used if *subset is not 'None*; "rand" selects the subset randomly, "first" uses the first *subset* images of the training data, and "last" uses the last *subset* images of the training data.

- **seed** (*int*) – only used if *subset* == *"rand"*; allows one to fix the random seed used to generate the subset (defaults to 1).
- **val_batch_size** (*None | int*) – if not *None*, specifies a different batch size for the validation set loader.
- **only_val** (*bool*) – If *True*, returns *None* in place of the training data loader
- **shuffle_train** (*bool*) – Whether or not to shuffle the training data in the returned DataLoader.
- **shuffle_val** (*bool*) – Whether or not to shuffle the test data in the returned DataLoader.

Returns

A training loader and validation loader according to the parameters given. These are standard PyTorch data loaders, and thus can just be used via:

```
>>> train_loader, val_loader = ds.make_loaders(workers=8, batch_
↪size=128)
>>> for im, lab in train_loader:
>>>     # Do stuff...
```

class robustness.datasets.**ImageNet**(*data_path*, ***kwargs*)

Bases: *robustness.datasets.DataSet*

ImageNet Dataset [DDS+09].

Requires ImageNet in ImageFolder-readable format. ImageNet can be downloaded from http://www.image-net.org. See here for more information about the format.

get_model(*arch*, *pretrained*)

class robustness.datasets.**Places365**(*data_path*, ***kwargs*)

Bases: *robustness.datasets.DataSet*

Places365 Dataset [ZLK+17], a 365-class scene recognition dataset.

See the places2 webpage for information on how to download this dataset.

get_model(*arch*, *pretrained*)

class robustness.datasets.**RestrictedImageNet**(*data_path*, ***kwargs*)

Bases: *robustness.datasets.DataSet*

RestrictedImagenet Dataset [TSE+19]

A subset of ImageNet with the following labels:

- Dog (classes 151-268)
- Cat (classes 281-285)
- Frog (classes 30-32)
- Turtle (classes 33-37)
- Bird (classes 80-100)
- Monkey (classes 365-382)
- Fish (classes 389-397)
- Crab (classes 118-121)
- Insect (classes 300-319)

To initialize, just provide the path to the full ImageNet dataset (no special formatting required).

get_model(*arch*, *pretrained*)

class robustness.datasets.**CustomImageNet**(*data_path*, *custom_grouping*, ***kwargs*)
Bases: *robustness.datasets.DataSet*

CustomImagenet Dataset

A subset of ImageNet with the user-specified labels

To initialize, just provide the path to the full ImageNet dataset along with a list of lists of wnids to be grouped together (no special formatting required).

get_model(*arch*, *pretrained*)

class robustness.datasets.**CIFAR**(*data_path='/tmp/'*, ***kwargs*)
Bases: *robustness.datasets.DataSet*

CIFAR-10 dataset [Kri09].

A dataset with 50k training images and 10k testing images, with the following classes:

- Airplane
- Automobile
- Bird
- Cat
- Deer
- Dog
- Frog
- Horse
- Ship
- Truck

get_model(*arch*, *pretrained*)

class robustness.datasets.**CINIC**(*data_path*, ***kwargs*)
Bases: *robustness.datasets.DataSet*

CINIC-10 dataset [DCA+18].

A dataset with the same classes as CIFAR-10, but with downscaled images from various matching ImageNet classes added in to increase the size of the dataset.

get_model(*arch*, *pretrained*)

class robustness.datasets.**A2B**(*data_path*, ***kwargs*)
Bases: *robustness.datasets.DataSet*

A-to-B datasets [ZPI+17]

A general class for image-to-image translation dataset. Currently supported are:

- Horse <-> Zebra
- Apple <-> Orange
- Summer <-> Winter

get_model(*arch*, *pretrained=False*)

class robustness.datasets.**OpenImages**(*data_path*, *custom_grouping=None*, ***kwargs*)
Bases: *robustness.datasets.DataSet*

OpenImages dataset [KDA+17]

More info: https://storage.googleapis.com/openimages/web/index.html

600-way classification with graular labels and bounding boxes.

..[KDA+17] Krasin I., Duerig T., Alldrin N., Ferrari V., Abu-El-Haija S., Kuznetsova A., Rom H., Uijlings J., Popov S., Kamali S., Malloci M., Pont-Tuset J., Veit A., Belongie S., Gomes V., Gupta A., Sun C., Chechik G., Cai D., Feng Z., Narayanan D., Murphy K. (2017). OpenImages: A public dataset for large-scale multi-label and multi-class image classification. Available from https://storage.googleapis.com/openimages/web/index.html.

get_model(*arch*, *pretrained*)

robustness.datasets.**DATASETS = {'a2b': <class 'robustness.datasets.A2B'>, 'cifar': <class**

```
>>> import robustness.datasets
>>> ds = datasets.DATASETS['cifar']('/path/to/cifar')
```

Type Dictionary of datasets. A dataset class can be accessed as

3.1.5 robustness.defaults module

This module is used to set up arguments and defaults. For information on how to use it, see Step 2 of the *Using robustness as a general training library (Part 1: Getting started)* walkthrough.

robustness.defaults.**TRAINING_DEFAULTS = {<class 'robustness.datasets.CIFAR'>: {'epochs':**
Default hyperparameters for training by dataset (tested for resnet50). Parameters can be accessed as *TRAINING_DEFAULTS[dataset_class][param_name]*

robustness.defaults.**TRAINING_ARGS = [['out-dir', <class 'str'>, 'where to save training log**
Arguments essential for the *train_model* function.

Format: *[NAME, TYPE/CHOICES, HELP STRING, DEFAULT (REQ=required, BY_DATASET=looked up in TRAINING_DEFAULTS at runtime)]*

robustness.defaults.**PGD_ARGS = [['attack-steps', <class 'int'>, 'number of steps for PGD at**
Arguments essential for the *robustness.train.train_model()* function if adversarially training or evaluating.

Format: *[NAME, TYPE/CHOICES, HELP STRING, DEFAULT (REQ=required, BY_DATASET=looked up in TRAINING_DEFAULTS at runtime)]*

robustness.defaults.**MODEL_LOADER_ARGS = [['dataset', ['imagenet', 'restricted_imagenet', '**
Arguments essential for constructing the model and dataloaders that will be fed into *robustness.train.train_model()* or *robustness.train.eval_model()*

Format: *[NAME, TYPE/CHOICES, HELP STRING, DEFAULT (REQ=required, BY_DATASET=looked up in TRAINING_DEFAULTS at runtime)]*

robustness.defaults.**CONFIG_ARGS = [['config-path', <class 'str'>, 'config path for loading**
Arguments for main.py specifically

Format: *[NAME, TYPE/CHOICES, HELP STRING, DEFAULT (REQ=required, BY_DATASET=looked up in TRAINING_DEFAULTS at runtime)]*

robustness.defaults.**add_args_to_parser**(*arg_list*, *parser*)

Adds arguments from one of the argument lists above to a passed-in arparse.ArgumentParser object. Formats helpstrings according to the defaults, but does NOT set the actual argparse defaults (*important*).

Parameters

- **arg_list** (*list*) – A list of the same format as the lists above, i.e. containing entries of the form [NAME, TYPE/CHOICES, HELP, DEFAULT]
- **parser** (*argparse.ArgumentParser*) – An ArgumentParser object to which the arguments will be added

Returns The original parser, now with the arguments added in.

robustness.defaults.**check_and_fill_args**(*args*, *arg_list*, *ds_class*)

Fills in defaults based on an arguments list (e.g., TRAINING_ARGS) and a dataset class (e.g., datasets.CIFAR).

Parameters

- **args** (*object*) – Any object subclass exposing `setattr` and `getattr` (e.g. cox.utils.Parameters)
- **arg_list** (*list*) – A list of the same format as the lists above, i.e. containing entries of the form [NAME, TYPE/CHOICES, HELP, DEFAULT]
- **ds_class** (*type*) – A dataset class name (i.e. a *robustness.datasets.DataSet* subclass name)

Returns The `args` object with all the defaults filled in according to `arg_list` defaults.

Return type args (object)

3.1.6 robustness.loaders module

robustness.loaders.**make_loaders**(*workers*, *batch_size*, *transforms*, *data_path*, *data_aug=True*, *custom_class=None*, *dataset=''*, *label_mapping=None*, *subset=None*, *subset_type='rand'*, *subset_start=0*, *val_batch_size=None*, *only_val=False*, *shuffle_train=True*, *shuffle_val=True*, *seed=1*, *custom_class_args=None*)

INTERNAL FUNCTION

This is an internal function that makes a loader for any dataset. You probably want to call dataset.make_loaders for a specific dataset, which only requires workers and batch_size. For example:

```
>>> cifar_dataset = CIFAR10('/path/to/cifar')
>>> train_loader, val_loader = cifar_dataset.make_loaders(workers=10, batch_
↪size=128)
>>> # train_loader and val_loader are just PyTorch dataloaders
```

class robustness.loaders.**PerEpochLoader**(*loader*, *func*, *do_tqdm=True*)

Bases: `object`

A blend between TransformedLoader and LambdaLoader: stores the whole loader in memory, but recomputes it from scratch every epoch, instead of just once at initialization.

compute_loader()

class robustness.loaders.**LambdaLoader**(*loader*, *func*)

Bases: `object`

This is a class that allows one to apply any given (fixed) transformation to the output from the loader in *real-time*.

For instance, you could use for applications such as custom data augmentation and adding image/label noise.

Note that the LambdaLoader is the final transformation that is applied to image-label pairs from the dataset as part of the loading process—i.e., other (standard) transformations such as data augmentation can only be applied *before* passing the data through the LambdaLoader.

For more information see *our detailed walkthrough*

Parameters

- **loader** (*PyTorch dataloader*) – loader for dataset (*required*).
- **func** (*function*) – fixed transformation to be applied to every batch in real-time (*required*). It takes in (images, labels) and returns (images, labels) of the same shape.

robustness.loaders.**TransformedLoader**(*loader*, *func*, *transforms*, *workers=None*, *batch_size=None*, *do_tqdm=False*, *augment=False*, *fraction=1.0*, *shuffle=True*)

This is a function that allows one to apply any given (fixed) transformation to the output from the loader *once*.

For instance, you could use for applications such as assigning random labels to all the images (before training).

The TransformedLoader also supports the application of addiotional transformations (such as standard data augmentation) after the fixed function.

For more information see *our detailed walkthrough*

Parameters

- **loader** (*PyTorch dataloader*) – loader for dataset
- **func** (*function*) – fixed transformation to be applied once. It takes
- **in** (*images, labels) and returns (images, labels*) –
- **every dimension except for the first, i.e., batch dimension** (*in*) –
- **can be any length)** (*(which*) –
- **transforms** (*torchvision.transforms*) – transforms to apply to the training images from the dataset (after func) (*required*).
- **workers** (*int*) – number of workers for data fetching (*required*).
- **batch_size** (*int*) – batch size for the data loaders (*required*).
- **do_tqdm** (*bool*) – if True, show a tqdm progress bar for the attack.
- **augment** (*bool*) – if True, the output loader contains both the original (untransformed), and new transformed image-label pairs.
- **fraction** (*float*) – fraction of image-label pairs in the output loader which are transformed. The remainder is just original image-label pairs from loader.
- **shuffle** (*bool*) – whether or not the resulting loader should shuffle every epoch (defaults to True)

Returns

A loader and validation loader according to the parameters given. These are standard PyTorch data loaders, and thus can just be used via:

```
>>> output_loader = ds.make_loaders(loader,
                                    assign_random_labels,
                                    workers=8,
                                    batch_size=128)
>>> for im, lab in output_loader:
>>>     # Do stuff...
```

3.1.7 robustness.main module

The main file, which exposes the robustness command-line tool, detailed in *this walkthrough*.

robustness.main.**main**(*args*, *store=None*)
Given arguments from *setup_args* and a store from *setup_store*, trains as a model. Check out the argparse object in this file for argument options.

robustness.main.**setup_args**(*args*)
Fill the args object with reasonable defaults from *robustness.defaults*, and also perform a sanity check to make sure no args are missing.

robustness.main.**setup_store_with_metadata**(*args*)
Sets up a store for training according to the arguments object. See the argparse object above for options.

3.1.8 robustness.model_utils module

class robustness.model_utils.**FeatureExtractor**(*submod*, *layers*)
Bases: sphinx.ext.autodoc.importer._MockObject

Tool for extracting layers from models.

Parameters

- **submod** (*torch.nn.Module*) – model to extract activations from
- **layers** (*list of functions*) – list of functions where each function, when applied to submod, returns a desired layer. For example, one function could be *lambda model: model.layer1*.

Returns

A model whose forward function returns the activations from the layers corresponding to the functions in *layers* (in the order that the functions were passed in the list).

forward(**args*, ***kwargs*)

class robustness.model_utils.**DummyModel**(*model*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*, **args*, ***kwargs*)

robustness.model_utils.**make_and_restore_model**(**_*, *arch*, *dataset*, *resume_path=None*, *parallel=False*, *pytorch_pretrained=False*, *add_custom_forward=False*)
Makes a model and (optionally) restores it from a checkpoint.

Parameters

- **arch** (*str|nn.Module*) – Model architecture identifier or otherwise a torch.nn.Module instance with the classifier

- **dataset** (*Dataset class [see datasets.py]*) –
- **resume_path** (*str*) – optional path to checkpoint saved with the robustness library (ignored if `arch` is not a string)
- **a string** (*not*) –
- **parallel** (*bool*) – if True, wrap the model in a DataParallel (defaults to False)
- **pytorch_pretrained** (*bool*) – if True, try to load a standard-trained checkpoint from the torchvision library (throw error if failed)
- **add_custom_forward** (*bool*) – ignored unless arch is an instance of nn.Module (and not a string). Normally, architectures should have a forward() function which accepts arguments `with_latent`, `fake_relu`, and `no_relu` to allow for adversarial manipulation (see 'here'<https://robustness.readthedocs.io/en/latest/example_usage/training_lib_part_2.html#training-with-custom-architectures> for more info). If this argument is True, then these options will not be passed to forward(). (Useful if you just want to train a model and don't care about these arguments, and are passing in an arch that you don't want to edit forward() for, e.g. a pretrained model)

Returns A tuple consisting of the model (possibly loaded with checkpoint), and the checkpoint itself

`robustness.model_utils.`**model_dataset_from_store**(*s*, *overwrite_params={}*, *which='last'*)

Given a store directory corresponding to a trained model, return the original model, dataset object, and args corresponding to the arguments.

3.1.9 robustness.train module

`robustness.train.`**check_required_args**(*args*, *eval_only=False*)

Check that the required training arguments are present.

Parameters

- **args** (*argparse object*) – the arguments to check
- **eval_only** (*bool*) – whether to check only the arguments for evaluation

`robustness.train.`**make_optimizer_and_schedule**(*args*, *model*, *checkpoint*, *params*)

Internal Function (called directly from train_model)

Creates an optimizer and a schedule for a given model, restoring from a checkpoint if it is non-null.

Parameters

- **args** (*object*) – an arguments object, see *train_model()* for details
- **model** (AttackerModel) – the model to create the optimizer for
- **checkpoint** (*dict*) – a loaded checkpoint saved by this library and loaded with *ch.load*
- **params** (*list|None*) – a list of parameters that should be updatable, all other params will not update. If `None`, update all params

Returns

An optimizer (ch.nn.optim.Optimizer) and a scheduler (ch.nn.optim.lr_schedulers module).

`robustness.train.`**eval_model**(*args*, *model*, *loader*, *store*)

Evaluate a model for standard (and optionally adversarial) accuracy.

Parameters

- **args** (*object*) – A list of arguments—should be a python object implementing `getattr()` and `setattr()`.
- **model** (`AttackerModel`) – model to evaluate
- **loader** (*iterable*) – a dataloader serving *(input, label)* batches from the validation set
- **store** (*cox.Store*) – store for saving results in (via tensorboardX)

robustness.train.**train_model**(*args*, *model*, *loaders*, *, *checkpoint=None*, *dp_device_ids=None*, *store=None*, *update_params=None*, *disable_no_grad=False*)

Main function for training a model.

Parameters

- **args** (*object*) – A python object for arguments, implementing `getattr()` and `setattr()` and having the following attributes. See *robustness.defaults.TRAINING_ARGS* for a list of arguments, and you can use *robustness.defaults.check_and_fill_args()* to make sure that all required arguments are filled and to fill missing args with reasonable defaults:

  adv_train (int or bool, *required*) if 1/True, adversarially train, otherwise if 0/False do standard training

  epochs (int, *required*) number of epochs to train for

  lr (float, *required*) learning rate for SGD optimizer

  weight_decay (float, *required*) weight decay for SGD optimizer

  momentum (float, *required*) momentum parameter for SGD optimizer

  step_lr (int) if given, drop learning rate by 10x every *step_lr* steps

  custom_lr_multplier (str)

  If given, use a custom LR schedule, formed by multiplying the original `lr` (format: [(epoch, LR_MULTIPLIER),...])

  lr_interpolation (str)

  How to drop the learning rate, either `step` or `linear`, ignored unless `custom_lr_multiplier` is provided.

  adv_eval (int or bool) If True/1, then also do adversarial evaluation, otherwise skip (ignored if adv_train is True)

  log_iters (int, *required*) How frequently (in epochs) to save training logs

  save_ckpt_iters (int, *required*) How frequently (in epochs) to save checkpoints (if -1, then only save latest and best ckpts)

  attack_lr (float or str, *required if adv_train or adv_eval*) float (or float-parseable string) for the adv attack step size

  constraint (str, *required if adv_train or adv_eval*) the type of adversary constraint (`robustness.attacker.STEPS`)

  eps (float or str, *required if adv_train or adv_eval*) float (or float-parseable string) for the adv attack budget

  attack_steps (int, *required if adv_train or adv_eval*) number of steps to take in adv attack

  custom_eps_multiplier (str, *required if adv_train or adv_eval*) If given, then set epsilon according to a schedule by multiplying the given eps value by a factor at each

epoch. Given in the same format as `custom_lr_multiplier`, `[(epoch, MULTIPLIER)..]`

use_best (int or bool, *required if adv_train or adv_eval*) : If True/1, use the best (in terms of loss) PGD step as the attack, if False/0 use the last step

random_restarts (int, *required if adv_train or adv_eval*) Number of random restarts to use for adversarial evaluation

custom_train_loss (function, optional) If given, a custom loss instead of the default CrossEntropyLoss. Takes in *(logits, targets)* and returns a scalar.

custom_adv_loss (function, *required if custom_train_loss*) If given, a custom loss function for the adversary. The custom loss function takes in *model, input, target* and should return a vector representing the loss for each element of the batch, as well as the classifier output.

custom_accuracy (function) If given, should be a function that takes in model outputs and model targets and outputs a top1 and top5 accuracy, will displayed instead of conventional accuracies

regularizer (function, optional) If given, this function of *model, input, target* returns a (scalar) that is added on to the training loss without being subject to adversarial attack

iteration_hook (function, optional) If given, this function is called every training iteration by the training loop (useful for custom logging). The function is given arguments *model, iteration #, loop_type [train/eval], current_batch_ims, current_batch_labels*.

epoch hook (function, optional) Similar to iteration_hook but called every epoch instead, and given arguments *model, log_info* where *log_info* is a dictionary with keys *epoch, nat_prec1, adv_prec1, nat_loss, adv_loss, train_prec1, train_loss*.

- **model** (`AttackerModel`) – the model to train.
- **loaders** (*tuple[iterable]*) – *tuple* of data loaders of the form *(train_loader, val_loader)*
- **checkpoint** (*dict*) – a loaded checkpoint previously saved by this library (if resuming from checkpoint)
- **dp_device_ids** (*list|None*) – if not `None`, a list of device ids to use for DataParallel.
- **store** (*cox.Store*) – a cox store for logging training progress
- **update_params** (*list*) – list of parameters to use for training, if None then all parameters in the model are used (useful for transfer learning)
- **disable_no_grad** (*bool*) – if True, then even model evaluation will be run with autograd enabled (otherwise it will be wrapped in a ch.no_grad())

3.1.10 robustness.tools package

Submodules

robustness.tools.constants module

robustness.tools.folder module

robustness.tools.folder.**has_file_allowed_extension**(*filename*, *extensions*)
Checks if a file is an allowed extension.

Parameters

- **filename** (*string*) – path to a file
- **extensions** (*iterable of strings*) – extensions to consider (lowercase)

Returns True if the filename ends with one of given extensions

Return type bool

robustness.tools.folder.**is_image_file**(*filename*)

Checks if a file is an allowed image extension.

Parameters **filename** (*string*) – path to a file

Returns True if the filename ends with a known image extension

Return type bool

robustness.tools.folder.**make_dataset**(*dir*, *class_to_idx*, *extensions*)

class robustness.tools.folder.**DatasetFolder**(*root*, *loader*, *extensions*, *transform=None*, *target_transform=None*, *label_mapping=None*)

Bases: sphinx.ext.autodoc.importer._MockObject

A generic data loader where the samples are arranged in this way:

```
root/class_x/xxx.ext
root/class_x/xxy.ext
root/class_x/xxz.ext

root/class_y/123.ext
root/class_y/nsdf3.ext
root/class_y/asd932_.ext
```

Parameters

- **root** (*string*) – Root directory path.
- **loader** (*callable*) – A function to load a sample given its path.
- **extensions** (*list[string]*) – A list of allowed extensions.
- **transform** (*callable, optional*) – A function/transform that takes in a sample and returns a transformed version. E.g, transforms.RandomCrop for images.
- **target_transform** – A function/transform that takes in the target and transforms it.

robustness.tools.folder.**pil_loader**(*path*)

robustness.tools.folder.**accimage_loader**(*path*)

robustness.tools.folder.**default_loader**(*path*)

class robustness.tools.folder.**ImageFolder**(*root*, *transform=None*, *target_transform=None*, *loader=<function default_loader>*, *label_mapping=None*)

Bases: *robustness.tools.folder.DatasetFolder*

A generic data loader where the images are arranged in this way:

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

Parameters

- **root** (*string*) – Root directory path.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.
- **loader** – A function to load an image given its path.

class robustness.tools.folder.**TensorDataset**(**tensors*, *transform=None*)
Bases: sphinx.ext.autodoc.importer._MockObject

Dataset wrapping tensors.

Each sample will be retrieved by indexing tensors along the first dimension.

Parameters ***tensors** (*Tensor*) – tensors that have the same size of the first dimension.

robustness.tools.helpers module

robustness.tools.helpers.**has_attr**(*obj*, *k*)
Checks both that obj.k exists and is not equal to None

robustness.tools.helpers.**calc_est_grad**(*func*, *x*, *y*, *rad*, *num_samples*)

robustness.tools.helpers.**ckpt_at_epoch**(*num*)

robustness.tools.helpers.**accuracy**(*output*, *target*, *topk=(1,)*, *exact=False*)
Computes the top-k accuracy for the specified values of k

Parameters

- **output** (*ch.tensor*) – model output (N, classes) or (N, attributes) for sigmoid/multitask binary classification
- **target** (*ch.tensor*) – correct labels (N,) [multiclass] or (N, attributes) [multitask binary]
- **topk** (*tuple*) – for each item "k" in this tuple, this method will return the top-k accuracy
- **exact** (*bool*) – whether to return aggregate statistics (if False) or per-example correctness (if True)

Returns A list of top-k accuracies.

class robustness.tools.helpers.**InputNormalize**(*new_mean*, *new_std*)
Bases: sphinx.ext.autodoc.importer._MockObject

A module (custom layer) for normalizing the input to have a fixed mean and standard deviation (user-specified).

forward(*x*)

class robustness.tools.helpers.**DataPrefetcher**(*loader*, *stop_after=None*)
Bases: object

preload()

class robustness.tools.helpers.**AverageMeter**
Bases: object

Computes and stores the average and current value

reset()

update(*val*, *n=1*)

robustness.tools.helpers.**get_label_mapping**(*dataset_name*, *ranges*)

robustness.tools.helpers.**restricted_label_mapping**(*classes*, *class_to_idx*, *ranges*)

robustness.tools.helpers.**custom_label_mapping**(*classes*, *class_to_idx*, *ranges*)

robustness.tools.label_maps module

robustness.tools.vis_tools module

robustness.tools.vis_tools.**get_axis**(*axarr*, *H*, *W*, *i*, *j*)

robustness.tools.vis_tools.**show_image_row**(*xlist*, *ylist=None*, *fontsize=12*, *size=(2.5*, *2.5)*, *tlist=None*, *filename=None*)

robustness.tools.vis_tools.**show_image_column**(*xlist*, *ylist=None*, *fontsize=12*, *size=(2.5*, *2.5)*, *tlist=None*, *filename=None*)

robustness.tools.vis_tools.**filter_data**(*metadata*, *criteria*, *value*)

robustness.tools.vis_tools.**plot_axis**(*ax*, *x*, *y*, *xlabel*, *ylabel*, ***kwargs*)

robustness.tools.vis_tools.**plot_tsne**(*x*, *y*, *npca=50*, *markersize=10*)

robustness.tools.breeds_helpers module

Module contents

CHAPTER 4

Contributors

- Andrew Ilyas
- Logan Engstrom
- Shibani Santurkar
- Dimitris Tsipras

Bibliography

[OMS17] Olah, et al., "Feature Visualization", Distill, 2017.

[IEA+18] Ilyas, A., Engstrom, L., Athalye, A., & Lin, J. (2018). Black-box adversarial attacks with limited queries and information. arXiv preprint arXiv:1804.08598.

[DDS+09] Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. 2009 IEEE Conference on Computer Vision and Pattern Recognition, 248-255.

[ZLK+17] Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., & Torralba, A. (2017). Places: A 10 million Image Database for Scene Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[TSE+19] Tsipras, D., Santurkar, S., Engstrom, L., Turner, A., & Madry, A. (2019). Robustness May Be at Odds with Accuracy. ICLR 2019.

[Kri09] Krizhevsky, A (2009). Learning Multiple Layers of Features from Tiny Images. Technical Report.

[DCA+18] Darlow L.N., Crowley E.J., Antoniou A., and A.J. Storkey (2018) CINIC-10 is not ImageNet or CIFAR-10. Report EDI-INF-ANC-1802 (arXiv:1810.03505)

[ZPI+17] Zhu, J., Park, T., Isola, P., & Efros, A.A. (2017). Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. 2017 IEEE International Conference on Computer Vision (ICCV), 2242-2251.

[EIS+19] Engstrom L., Ilyas A., Santurkar S., Tsipras D., Tran B., Madry A. (2019). Learning Perceptually-Aligned Representations via Adversarial Robustness. arXiv, arXiv:1906.00945

[STE+19] Santurkar S., Tsipras D., Tran B., Ilyas A., Engstrom L., Madry A. (2019). Image Synthesis with a Single (Robust) Classifier. arXiv, arXiv:1906.09453

[STM20] Santurkar S., Tsipras D., Madry A. (2020). : BREEDS: Benchmarks for Subpopulation Shift. arXiv, arXiv:2008.04859

Python Module Index

r

Index

H

I

L

M

O

P

R

S

T

U