Robustness Gym

Release v0.1.2

Robustness Gym

Aug 26, 2021

GETTING STARTED

Robustness Gym is a toolkit for evaluating natural language processing models.

Robustness Gym is *under active development* so expect rough edges. Feedback and contributions are welcomed and appreciated. You can submit bugs and feature suggestions on Github Issues and submit contributions using a pull request.

You can get started by going to the *installation* page.

CHAPTER

ONE

INSTALLATION

This page describes how to get Robustness Gym installed and ready to use. Head to the tutorials to start using Robustness Gym after installation.

1.1 Installing the Robustness Gym package

The only things you need to install to get setup.

1.1.1 Install with pip

```
pip install robustnessgym
```

1.2 Optional Installation

The steps below aren't necessary unless you need these features.

1.2.1 Progress bars in Jupyter

Enable the following Jupyter extensions to display progress bars properly.

```
jupyter nbextension enable --py widgetsnbextension
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

1.2.2 TextBlob setup

To use TextBlob, download and install the TextBlob corpora.

```
python -m textblob.download_corpora
```

1.2.3 Installing Spacy GPU

To install Spacy with GPU support, use the installation steps given below.

```
pip install cupy
pip install spacy[cuda]
python -m spacy download en_core_web_sm
```

1.2.4 Installing neuralcoref

The standard version of `neuralcoref` does not use GPUs for prediction and a pull request that is pending adds this functionality. Follow the steps below to use this.

```
git clone https://github.com/dirkgr/neuralcoref.
↪git@754d470d484f56c5715ef35c220c217f28079eef
cd neuralcoref
git checkout GpuFix
pip install -r requirements.txt
pip install -e .
```

CHAPTER

TWO

ROBUSTNESS GYM IN A NUTSHELL

What is Robustness Gym? Should you use it? Read this page to find some quick answers to common questions.

2.1 The Big Picture

Robustness Gym was built out of our own frustrations of being unable to systematically evaluate and test our machine learning models.

Traditionally, evaluation has consisted of a few simple steps:

1. Load some data
2. Generate predictions using a model
3. Compute aggregate metrics

This is no longer sufficient: models are increasingly being deployed in real-world use cases, and aggregate performance is too coarse to make meaningful model assessments. Modern evaluation is about understanding if models are *robust* to all the scenarios they might encounter, and where the tradeoffs lie.

This is reflected in Robustness Gym which distills these modern goals into a new workflow,

1. Load some data
2. Compute and cache side-information on data
3. Build slices of data
4. Evaluate across the slices
5. Report and share findings
6. Iterate

We'll go into what these steps mean and how to use them in Robustness Gym next.

2.2 The Robustness Gym Workflow

2.2.1 1. Load some data

Loading data in Robustness Gym is easy. We extend the Huggingface datasets library, so all datasets there are immediately available for use using the Robustness Gym `Dataset` class.

```
import robustnessgym as rg

# Load the boolq data
dataset = rg.Dataset.load_dataset('boolq', split='train[:10]')

# Load the first 10 training examples
dataset = rg.Dataset.load_dataset('boolq', split='train[:10]')

# Load from jsonl file
dataset = rg.Dataset.from_json("file.jsonl")
```

2.2.2 2. Compute and cache side-information

One of the most common operations in evaluation is interpreting and analyzing examples in dataset. This can mean tagging data, adding additional information about examples from a knowledge base, or making predictions about the example.

It's often useful to have this information available conveniently stored alongside the example, ready to use for analysis.

This is the idea of the `CachedOperation` class in Robustness Gym. Think of it as a `.map()` over your dataset, except it provides convenience functions to retrieve any information you cache.

Robustness Gym ships with a few cached operations that you can use out-of-the-box.

```
from robustnessgym import SpacyOp, Stanza, TextBlob

# Create the Spacy CachedOperation
spacy_op = SpacyOp()

# Apply it on the "text" column of a dataset
dataset = spacy_op(batch_or_dataset=dataset, columns=["text"])

# Easily retrieve whatever information you need, wherever you need it

# Retrieve the tokens extracted by Spacy for the first 2 examples in the dataset
tokens = SpacyOp.retrieve(batch=dataset[:2], columns=["text"], proc_fns=SpacyOp.tokens)

# Retrieve everything Spacy cached for the first 2 examples, and process it yourself
spacy_info = SpacyOp.retrieve(batch=dataset[:2], columns=["text"])

# ...do stuff with spacy_info
```

2.2.3 3. Build slices

Robustness Gym supports a general set of abstractions to create slices of data. Slices are just datasets that are constructed by applying an instance of the `SliceBuilder` class in Robustness Gym.

Robustness Gym currently supports slices of four kinds:

1. **Evaluation Sets**: slice constructed from a pre-existing dataset
2. **Subpopulations**: slice constructed by filtering a larger dataset
3. **Transformations**: slice constructed by transforming a dataset

4. **Attacks**: slice constructed by attacking a dataset adversarially

3.1 Evaluation Sets

```
from robustnessgym import Dataset, Slice

# Evaluation Sets: direct construction of a slice
boolq_slice = Slice(Dataset.load_dataset('boolq'))
```

3.2 Subpopulations

```
from robustnessgym import NumTokensSubpopulation

# A simple subpopulation that splits the dataset into 3 slices
# The intervals act as buckets: the first slice will bucket based on text with
# length between 0 and 4
length_sp = NumTokensSubpopulation(intervals=[(0, 4), (8, 12), ("80%", "100%")])

# Apply it
dataset, slices, membership = length_sp(batch_or_dataset=dataset, columns=['text'])

# dataset is an updated dataset where every example is tagged with its slice
# slices are a list of Slice objects: think of this as a list of 3 datasets
# membership is a matrix of shape (n x 3) with 0/1 entries, assigning each of the n
# examples depending on whether they're in the slice or not
```

3.3 Transformations

```
from robustnessgym import EasyDataAugmentation

# Easy Data Augmentation (https://github.com/jasonwei20/eda_nlp)
eda = EasyDataAugmentation(num_transformed=2)

# Apply it
dataset, eda_slices, eda_membership = eda(batch_or_dataset=dataset, columns=['text'])

# eda_slices is just 2 transformed versions of the original dataset
```

3.4 Attacks

```
from robustnessgym import TextAttack
from textattack.models.wrappers import HuggingFaceModelWrapper

# TextAttack
textattack = TextAttack.from_recipe(recipe='BAEGarg2019',
                                    model=HuggingFaceModelWrapper(...))
```

2.2.4 4. Evaluate slices

At this point, you can just use your own code (e.g. in numpy) to calculate metrics , since the slices are just datasets.

```
import numpy as np

def accuracy(true: np.array, pred: np.array):
    """
    Your function for computing accuracy.
    """
    return np.mean(true == pred)

# Some model in your code
model = MyModel()

# Evaluation on the length slices
metrics = {}
for sl in slices:
    metrics[sl.identifier] = accuracy(true=sl["label"], pred=MyModel.predict(sl['text']))
```

Robustness Gym includes a `TestBench` abstraction to make this process easier.

```
from robustnessgym import TestBench, Identifier, BinarySentiment

# Construct a testbench
testbench = TestBench(
    # Your identifier for the testbench
    identifier=Identifier(_name="MyTestBench"),
    # The task this testbench should be used to evaluate
    task=BinarySentiment(),
)

# Add slices
testbench.add_slices(slices)

# Evaluate: Robustness Gym knows what metrics to use from the task
metrics = testbench.evaluate(model)
```

You can also get a Robustness Report using the TestBench.

```
# Create the report
report = testbench.create_report(model)

# Generate the figures
_, figure = report.figures()
figure.write_image('my_figure.pdf', engine="kaleido")
```

CHAPTER

THREE

QUICKSTART

This page gives a quick overview on how to start using Robustness Gym.

The central operation in Robustness Gym is the construction of *slices* of data: a slice is just a dataset that is used to test specific model properties.

Robustness Gym comes with a set of general abstractions to build slices with ease. We'll use a simple example to show you how these work.

Robustness Gym also has a lot of built-in functionality that you can use out-of-the-box (thanks to some other great open-source projects) for creating slices. You can read more about these in [](), and check out []() if you'd like to contribute some of your own slice building code to Robustness Gym.

Let's dive in quickly!

3.1 Building Slices

Robustness Gym contains a `SliceBuilder` class for writing code to build slices. This class defines a common interface that all `SliceBuilders` must follow:

1. Any `SliceBuilder` object can be called using `slicebuilder(batch_or_dataset, columns)`.
2. This call always returns a `(dataset, slices, matrix)` tuple.

To see how this works, let's see a simple example. We're going to

1. Create a dummy dataset containing just 4 text examples.
2. Use a `ScoreSubpopulation` (a kind of `SliceBuilder`) to build 2 slices.

Let's start by creating the dataset.

```
from robustnessgym import Dataset, Identifier

dataset = Dataset.from_batch({
    'text': ['a person is walking',
             'a person is running',
             'a person is sitting',
             'a person is walking on a street eating a bagel']
}, identifier=Identifier(_name='MyDataset'))
```

Here, we used the `.from_batch(..)` method to create a dataset called `MyDataset`. This dataset has a single column called *text* with 4 examples or rows.

The `Identifier` class is used to store identifying information for `Dataset` objects, `SliceBuilder` objects and more.

Tip: Most objects in Robustness Gym have a `.identifier` property that can be used to inspect the object.

Next, let's create the `ScoreSubpopulation` to build slices.

```
def length(batch, columns):
    """
    A simple function to compute the length of all examples in a batch.

    batch: a dict of lists
    columns: a list of str

    return: a list of lengths
    """
    assert len(columns) == 1, "Pass in a single column."

    # The name of the column to grab text from
    column_name = columns[0]
    text_batch = batch[column_name]

    # Tokenize the text using .split() and calculate the number of tokens
    return [len(text.split()) for text in text_batch]
```

We pause here to point out three things:

1. The `def func(batch, columns)` is a common pattern in Robustness Gym for adding custom functionality.

   The `batch` here refers to a batch of data,

   ```
   {'text': ['a person is walking', 'a person is running'], 'index': [0, 1]}
   ```

   is a batch of size 2 from the dataset (`dataset[:2]`).

   The `columns` parameter specifies the relevant columns of the batch. This has some advantages e.g. suppose `otherdataset` has a column of text named *sentence* instead. We can reuse `length` for both datasets,

   ```
   length(batch=dataset[:2], columns=['text'])
   length(batch=otherdataset[:2], columns=['sentence'])
   ```

2. `length` returns a list of scores (lengths in this case). This is an important ingredient of the `ScoreSubpopulation`, which constructs (as the name suggests) slices by bucketing examples based on their score.

3. We tokenized text inside the length function. This is bad:

   1. Tokenization is a basic step in text processing, and we should only do it once.
   2. If it was some other, more expensive operation, we should definitely do it once.

Let's keep going and wrap `length` in a `ScoreSubpopulation`.

```
from robustnessgym import ScoreSubpopulation

# Create the score subpopulation for length
length_sp = ScoreSubpopulation(intervals=[(0, 5), (5, 10)], score_fn=length)
```

The `ScoreSubpopulation` requires

1. a list of `intervals`, each interval is a tuple containing the range of lengths that are considered part of that slice.
2. a `score_fn`, used to assign scores to a batch of examples

Let's run this on the dataset.

```
# Run the length subpopulation on the dataset
dataset, slices, membership = length_sp(batch_or_dataset=dataset, columns=['text'])
```

This call just executes the `length` function on the dataset, and buckets the examples based on which intervals they fall in. As we briefly mentioned earlier, this returns the `(dataset, slices, membership)` tuple,

1. `dataset` now tags each example with slice information i.e. what slices does the example belong to
2. `slices` is a list of `Slice` objects (2 here, since we specified 2 intervals). Each `Slice` object is a dataset containing just the examples that were part of the slice.
3. `membership` is a `np.array` matrix of shape `(n, m)`, where `n` is the number of examples in the original dataset, and `m` is the number of slices built. Entry `(i, j)` is 1 if example `i` is in slice `j`.

And that's (almost) it! Most code you write in Robustness Gym will follow a similar workflow. Before we end, we take a short segue to talk about the other major abstraction in Robustness Gym: the `CachedOperation` class.

3.2 Caching Information

As we noted earlier, we tokenized text inside the `length` function, when we should ideally run this step separately and reuse it across multiple `SliceBuilder` objects.

When creating Robustness Gym, we noticed this pattern frequently: cache some information (`CachedOperation`), and use that information to build some slices (`SliceBuilder`).

Let's look at the same example as before, and use a `CachedOperation` for tokenization this time.

```
from robustnessgym import CachedOperation, Identifier

def tokenize(batch, columns):
    """
    A simple function to tokenize a batch of examples.

    batch: a dict of lists
    columns: a list of str

    return: a list of tokenized text
    """
    assert len(columns) == 1, "Pass in a single column."

    # The name of the column to grab text from
    column_name = columns[0]
    text_batch = batch[column_name]

    # Tokenize the text using .split()
    return [text.split() for text in text_batch]

# Create the CachedOperation
cachedop = CachedOperation(apply_fn=tokenize,
                           identifier=Identifier(_name="Tokenizer"))
```

We've written `tokenize` with the familiar `func(batch, columns)` function signature. This function is then wrapped into a `CachedOperation` for use.

Tip: A `CachedOperation` can be created with *any* `func(batch, columns)`. The only constraint is that it must return a list, with size equal to that of the batch.

Let's create our `ScoreSubpopulation` for length again.

```
from robustnessgym.decorators import singlecolumn

def length(batch, columns):
    """
    A simple function to compute the length of all examples in a batch.

    batch: a dict of lists
    columns: a list of str

    return: a list of lengths
    """
    assert len(columns) == 1, "Pass in a single column."

    # The name of the column to grab text from
    column_name = columns[0]
    text_batch = batch[column_name]

    CachedOperation.retrieve(
        batch=batch,
        columns=[column_name],
        proc_fns=lambda decoded_batch: []
    )

    # Tokenize the text using .split() and calculate the number of tokens
    return [len(text.split()) for text in text_batch]
```

Robustness Gym ships with `CachedOperations` that use standard text processing pipelines to tokenize and tag text.

There's a ton more to Robustness Gym (and more coming). Here are some pointers on where to head to next, depending on your specific goals:

1. If you want a more detailed tutorial and walkthrough, head to the [Tutorial 1]() Jupyter notebook
2. If you'd like to see what `SliceBuilders` are available in Robustness Gym today, check out []().
3. If you're interested in a walkthrough of the `SliceBuilder` class in more detail, head to [](). Head to []() for a deep dive into the `CachedOperation` class. This is recommended for expert users.
4. If you'd like to learn more about the motivation behind Robustness Gym, check out []().
5. If you're interested in becoming a contributor, read []().

CHAPTER

FOUR

API REFERENCE

4.1 robustnessgym.active package

4.1.1 Submodules

4.1.2 robustnessgym.active.ais module

4.1.3 robustnessgym.active.mandoline module

4.1.4 Module contents

4.2 robustnessgym.core package

4.2.1 Submodules

4.2.2 robustnessgym.core.constants module

4.2.3 robustnessgym.core.dataset_to_task module

4.2.4 robustnessgym.core.decorators module

4.2.5 robustnessgym.core.devbench module

4.2.6 robustnessgym.core.identifier module

4.2.7 robustnessgym.core.metrics module

4.2.8 robustnessgym.core.model module

4.2.9 robustnessgym.core.operation module

4.2.10 robustnessgym.core.slice module

4.2.11 robustnessgym.core.storage module

4.2.12 robustnessgym.core.testbench module

4.2.13 robustnessgym.core.tools module

4.2.14 robustnessgym.core.version module

4.2.15 Module contents

robustnessgym.slicebuilders.transformations.similarity module

Module contents

4.8.2 Submodules

4.8.3 robustnessgym.slicebuilders.attack module

4.8.4 robustnessgym.slicebuilders.slicebuilder module

4.8.5 robustnessgym.slicebuilders.slicebuilder_collection module

4.8.6 robustnessgym.slicebuilders.subpopulation module

4.8.7 robustnessgym.slicebuilders.subpopulation_collection module

4.8.8 robustnessgym.slicebuilders.transformation module

4.8.9 Module contents

4.9 robustnessgym.tasks package

4.9.1 Submodules

4.9.2 robustnessgym.tasks.schema module

4.9.3 robustnessgym.tasks.task module

4.9.4 Module contents