Empirical Investigation of Representation Learning for Imitation (EIRLI)

Release 0.1.0

Center for Human-Compatible AI

Mar 04, 2023

CONTENTS:

Over the past handful of years, representation learning has exploded as a subfield, and, with it have come a plethora of new methods, each slightly different from the other.

Our Empirical Investigation of Representation Learning for Imitation (EIRLI) has two main goals:

1. To create a modular algorithm definition system that allows researchers to easily pick and choose from a wide array of commonly used design axes
2. To facilitate testing of representations within the context of sequential learning, particularly imitation learning and offline reinforcement learning

CHAPTER

ONE

COMMON USE CASES

Do you want to…

- Reproduce our results? You can find scripts and instructions here to help reproduce our benchmark results.
- Design and experiment with a new representation learning algorithm using our modular components? You can find documentation on that here
- Use our algorithm definitions in a setting other than sequential learning? The base example here demonstrates this simplified use case

Otherwise, you can see our full ReadTheDocs documentation here.

CHAPTER

TWO

MODULAR ALGORITHM DESIGN

This library was designed in a way that breaks down the definition of a representation learning algorithm into several key parts. The intention was that this system be flexible enough many commonly used algorithms can be defined through different combinations of these modular components.

The design relies on the central concept of a "context" and a "target". In very rough terms, all of our algorithms work by applying some transformation to the context, some transformation to the target, and then calculating a loss as a function of those two transformations. Sometimes an extra context object is passed in

Some examples are:

- In **SimCLR**, the context and target are the same image frame, and augmentation and then encoding is applied to both context and target. That learned representation is sent through a decoder, and then the context and target representations are pulled together with a contrastive loss.
- In **TemporalCPC**, the context is a frame at time t, and the target a frame at time t+k, and then, similarly to SimCLR above, augmentation is applied to the frame before it's put through an encoder, and the two resulting representations pulled together
- In a **Variational Autoencoder**, the context and target are the same image frame. An bottleneck encoder and then a reconstructive decoder are applied to the context, and this reconstructed context is compared to the target through a L2 pixel loss
- A **Dynamics Prediction** model can be seen as an conceptual combination of an autoencoder (which tries to predict the current full image frame) and TemporalCPC, which predicts future information based on current information. In the case of a Dynamics model, we predict a future frame (the target) given the current frame (context) and an action as extra context.

This abstraction isn't perfect, but we believe it is coherent enough to allow for a good number of shared mechanisms between algorithms, and flexible enough to support a wide variety of them.

The modular design mentioned above is facilitated through the use of a number of class interfaces, each of which handles a different component of the algorithm. By selecting different implementations of these shared interfaces, and creating a RepresentationLearner that takes them as arguments, and handles the base machinery of performing transformations.

1. TargetPairConstructer - This component takes in a set of trajectories (assumed to be iterators of dicts containing 'obs' and optional 'acts', and 'dones' keys) and creates a dataset of (context, target, optional extra context) pairs that will be shuffled to form the training set.
2. Augmenter - This component governs whether either or both of the context and target objects are augmented before being passed to the encoder. Note that this concept only meaningfully applies when the object being augmented is an image frame.
3. Encoder - The encoder is responsible for taking in an image frame and producing a learned vector representation. It is optionally chained with a Decoder to produce the input to the loss function (which may be a reconstructed

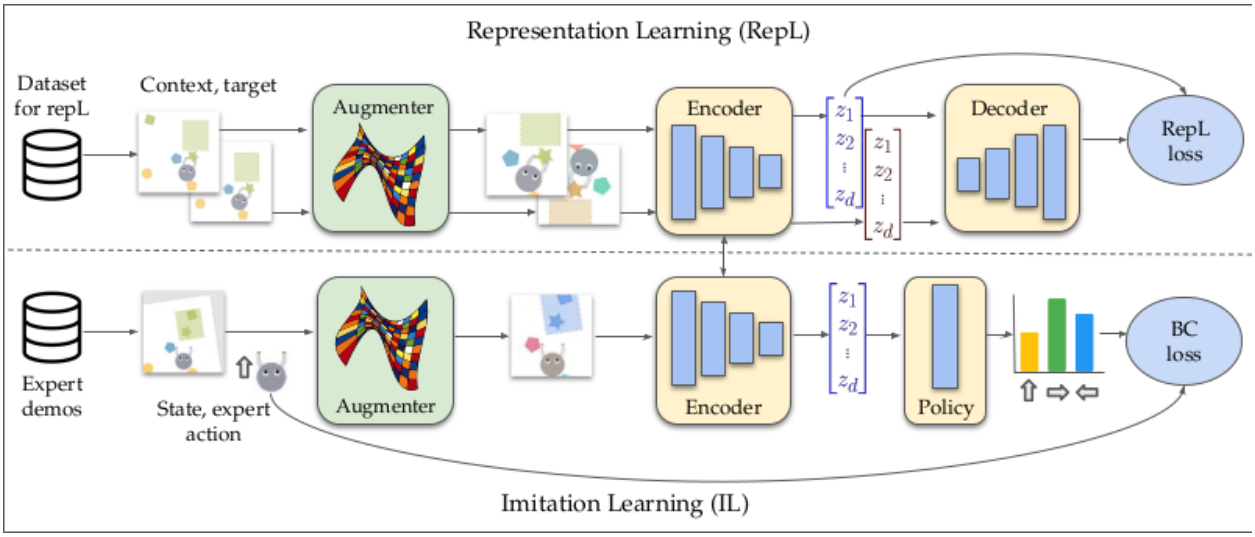

Fig. 1: A diagram showing how these components made up a training pipeline for our benchmark

image in the case of VAE or Dynamics, or may be a projected version of the learned representation in the case of contrastive methods like SimCLR that use a projection head)

4. Decoder - As mentioned above, the Decoder acts as a bridge between the representation in the form you want to use for transfer, and whatever input is required your loss function, which is often some transformation of that canonical representation.

5. BatchExtender - This component is used for situations where you want to calculate loss on batch elements that are not part of the batch that went through your encoder and decoder on this step. This is centrally used for contrastive methods that use momentum, since in that case, you want to use elements from a cached store of previously-calculated representations as negatives in your contrastive loss

6. LossCalculator - This component takes in the transformed context and transformed target and handles the loss calculation, along with any transformations that need to happen as a part of that calculation.

CHAPTER

THREE

TRAINING SCRIPTS

In addition to machinery for constructing algorithms, the repo contains a set of Sacred-based training scripts for testing different Representation Learning algorithms as either pretraining or joint training components within an imitation learning pipeline. These are likeliest to be a fit for your use case if you want to reproduce our results, or train models in similar settings

3.1 Reproduction of Benchmark Paper Experiments

3.1.1 Getting data and setting up an output directory for runs

To reproduce our results, you will first need to download and extract our demonstration dataset (around 8GiB). We'll assume that the extracted dataset directory is at `/path/to/extracted/data`.

You'll also want to create a new directory to store the results of your runs. This will need to hold ~200GiB if you run all the experiments (although it won't need as much space if you only run one experiment at a time and delete output files as you go). We'll assume that this directory is at `/path/to/runs/dir`.

3.1.2 Building the code

The easiest way to use our code is as a Docker image. We have a script that takes a snapshot of the git repository and bakes it into a Docker image with the `Dockerfile` in the root of this repo. The script can also set the UID and GID inside the container to match the user's UID and GID on the host machine, so that files written from inside the container will be owned by the user on the host machine. To run the script, use the following command:

```
cd /path/to/eirli-git-repository  # change the path
# -u and -n control the UID and the username inside the container, respectively.
# -d controls the start of the image name (e.g. -d foobar results in an
# image calleed "foobar/<something>"). You can change this to your Docker
# Hub username if you want to push to Docker Hub later on.
./cloud/build_docker.sh -u $UID -n $USER -d docker-hub-user
```

This may take a few minutes because it needs to install all the system-level and Python-level dependencies in the Docker image. The final Docker image will be about 17GiB.

You can verify that the Docker image was built correctly by running the following command:

```
docker run -it --rm docker-hub-user/eirli-$USER:latest \
    bash -c 'echo "Hello, world!" && pkill -9 ray'
```

This will start up an instance of Ray (a distributed computing framework) and an X server (for environment rendering) inside the container. This will produce some messages and warnings. About 30 seconds after that, it will print "Hello, world!" and then exit.

3.1.3 Running the code

Once you've built a Docker image, you launch experiments in new Docker containers:

```
# configuration variables
# (CHANGE PATHS!)
path_to_extracted_data=/path/to/extracted/data
path_to_store_runs=/path/to/runs/dir
image_name=docker-hub-user/eirli-$USER:latest
cpus=24
memory=100g
# can also use, e.g., device=1,2 or device=3 or device=0,3,4 etc.
# (if using more GPUs, you should increase the CPU and memory limits
# proportionally)
gpus="device=0"

run_in_docker() {
    docker run -it --rm \
        --cpus="$cpus" --memory="$memory" --gpus="$gpus" --shm-size=15g \
        --volume "$(realpath "$path_to_extracted_data"):/data:ro" \
        --volume "$(realpath "$path_to_store_runs"):/runs:rw" \
        "$image_name" "$@"
}

# Joint training experiments (dm_control, MAGICAL, Procgen)
run_in_docker bc_jt_expts_dmc.sh /data /runs
run_in_docker bc_jt_expts_magical.sh /data /runs
run_in_docker bc_jt_expts_procgen.sh /data /runs

# BC + pretrained repL experiments (dm_control, MAGICAL, Procgen)
run_in_docker bc_pretrain_expts_dmc.sh /data /runs
run_in_docker bc_pretrain_expts_magical.sh /data /runs
run_in_docker bc_pretrain_expts_procgen.sh /data /runs

# GAIL + pretrained repL experiments (dm_control, MAGICAL, Procgen)
run_in_docker gail_expts_dmc.sh /data /runs
run_in_docker gail_expts_magical.sh /data /runs
run_in_docker gail_expts_procgen.sh /data /runs
```

On one A6000 GPU, each of these scripts will take somewhere between a few days (for the GAIL experiments) and a couple of weeks (BC pretraining experiments) to complete.

3.1.3.1 Read this if you have less than 40GiB of VRAM

These scripts were tuned to run on GPUs with 40GiB+ of memory, such as the A6000 or the 40G A100. If you have a GPU with less VRAM then you might run out of memory, since each script runs 10+ jobs in parallel on each GPU. To run fewer experiments per GPU, you can edit the job launch scripts in `cloud/` (e.g. `bc_jt_expts_dmc.sh`). The specific section you need to edit looks like this:

```
gpu_default=0.11
declare -A gpu_overrides=(
    ["repl_tcpc8_192"]="0.16"
    ["repl_simclr_192"]="0.16"
)
```

These variables indicate what fraction of the GPU memory to use for each job (with overrides for some representation learning algorithms that use more memory). You can increase these fractions to decrease the number of jobs per GPU. Once you're done, you'll need to rebuild the Docker image and re-run the above commands (the rebuild should be much faster because everything except the EIRLI source code will have been cached by Docker).

3.2 Dataset Creation & Environment Specification

This document explains the abstractions that we are using to load demonstration data and create Gym environments. These abstractions are intended provide a reasonably uniform internal interface across all of all the benchmarks supported by the il-representations project (Atari, dm_control, MAGICAL, Minecraft, etc.).

3.2.1 Sacred ingredients

The data-loading pipeline is configured using three different Sacred ingredients:

- env_cfg_ingredient: In principle, this ingredient contains all the information necessary to create a Gym environment for a specific combination of benchmark and task. The two most important config keys are `benchmark_name` (which identifies whether the current benchmark is MAGICAL, or dm_control, or something else), and `task_name` (which identifies the current task within the selected benchmark; e.g. finger-spin or MoveToCorner-Demo-v0). There are also some benchmark-specific config keys for, e.g., preprocessing.
- venv_opts_ingredient: Additional options required to construct a vecenv (e.g. the number of environments to run in parallel).
- env_data_ingredient: Contains paths to data files on disk. Has quite a few dataset-specific keys, particularly for loading 'native'-format datasets (as described further down).

Not every script requires every one of the above ingredients. For instance, testing a trained policy requires `env_cfg_ingredient` to determine which environment to evaluate on, and `venv_opts_ingredient` to construct a vecenv, but not `env_data_ingredient`. As a result, the three components have been separated out to minimise the number of redundant Sacred config options for each script.

3.2.2 Creating Gym environments

Gym environments can be created with auto.load_vec_env(). This uses `env_cfg['benchmark_name']` (from the `env_cfg_ingredient` Sacred ingredient) to dispatch to a benchmark-specific routine for creating vecenvs. The benchmark-specific routines make use of both `env_cfg['task_name']` and (possibly) some benchmark-specific keys in `env_cfg` to, e.g., apply appropriate preprocessors. In addition, `auto.load_vec_env()` uses `venv_opts` (from the `venv_opts_ingredient` Sacred ingredient) to determine, e.g., how many environments the vecenv should run in parallel.

3.2.3 Loading demonstrations from their 'native' format

Demonstrations for each benchmark were originally generated in a few different formats. For instance, the MAGICAL reference demonstrations are distributed as pickles, while the Atari demonstrations were saved as Numpy `.npz` files (see the data formats GDoc for more detail). The auto.load_dataset_dict() function provides a uniform interface to these formats.

Like `auto.load_vec_env()`, the `auto.load_dict_dataset()` function uses `env_cfg['benchmark_name']` to dispatch to a benchmark-specific data-loading function that is able to read the benchmark's on-disk data format. Those benchmark-specific loading functions in turn look at benchmark-specific config keys in `env_data` (from `env_data_ingredient`) to locate the demonstrations. For example, `benchmark_name="magical"` dispatches to envs.magical_envs.load_data(), which looks up the current task name (i.e. `env_cfg["task_name"]`) in `env_data["magical_demo_dirs"]` to determine where the relevant demonstrations are stored.

Regardless of the value of `env_cfg['benchmark_name']`, `auto.load_dataset_dict()` always returns a dict with the following keys:

- `obs`: an `N*C*H*W` array of observations associated with states.
- `next_obs`: an `N*C*H*W` array of observations associated with the state *after* the corresponding one in `obs`.
- `acts`: an `N*A_1*A_2*...` array of actions, where `A_1*A_2*...` is the shape of the action space.
- `dones`: an length-`N` array of bools indicating whether the corresponding state was terminal.

Note that `N` here is the sum of the lengths of all trajectories in the dataset; trajectories are concatenated together to form each value in the returned dictionary. It is possible to segment the values back into trajectories by looking at the `dones` array.

Loading all demonstrations into a single dictionary in memory has one major advantage, but also a few drawbacks. The advantage is that it's easy to manipulate the demonstrations: you can figure out how many trajectories you have with `np.sum(data_dict["dones"])`, or randomly index into time steps in order to construct shuffled batches. The three main disadvantages are:

- Loading all demonstrations into a dict can use a lot of memory.
- `auto.load_dict_dataset()` relies on the `env_cfg_ingredient` Sacred ingredient, which only supports specifying a single training task. Thus it is not easy to extend `auto.load_dict_dataset()` so that it can load multitask data.
- It's hard to invert `auto.load_dict_dataset()` into a function for *saving* trajectories, since it needs to support several different (benchmark-specific) data formats. However, it would be convenient to have such an inverse function, since that would allow us to write benchmark-agnostic code for generating new repL training data (e.g. generating training data from random rollouts).

For these reasons, we also have a second data format…

3.2.4 The webdataset format

In addition to the in-memory dict format generated by `auto.load_dict_dataset()`, we also have a second set of independent data-saving and data-loading machinery based on the webdataset spec/library. This section briefly explains how webdataset works, and how we use it to load data for the `run_rep_learner` script.

3.2.4.1 High-level interface and configuration

Within our codebase, the high-level interface for loading datasets in the webdataset format is the auto.load_wds_datasets() function. This takes a list of configurations for single-task datasets, and returns an list containing one webdataset `Dataset` for each task. It is then the responsibility of the calling code to apply any necessary preprocessing steps to those `Dataset`s (e.g. target pair construction) and to multiplex the datasets with an `InterleavedDataset`. These abstractions are explained further down the page.

The configuration syntax for `auto.load_wds_datasets()` is exactly the syntax used for the `dataset_configs` configuration option in `run_rep_learner.py`, and as such deserves some further explanation. Each element of the list passed to `auto.load_wds_datasets()` is a dict which may contain the following keys:

```
{
    # the type of data to be loaded
    "type": "demos" | "random" |
    # a dictionary containing some subset of configuration keys from `env_cfg_ingredient`
    "env_cfg": {...},
}
```

Both the `"type"` key and the `"env_cfg"` key are optional. `"type"` defaults to `"demos"`, and `"env_cfg"` defaults to the current configuration of `env_cfg_ingredient`. If any sub-keys are provided in `"env_cfg"`, then they are recursively combined with the current configuration of `"env_cfg_ingredient"`. This allows one to define new dataset configurations that override only some aspects of the current `"env_cfg_ingredient"` configuration.

This configuration syntax might be clearer with a few examples:

- Training on random rollouts and demonstrations using the current benchmark name from `env_cfg_ingredient`:

```
dataset_configs = [{"type": "demos"}, {"type": "random"}]
```

- Training on demos from both the default task from `env_cfg_ingredient`, and another task called "finger-spin". Notice that this time the first config dict does not have *any* keys; this is equivalent to using `{"type": "demos"}` as we did above. `"type": "demos"` is also implicit in the second dict.

```
dataset_configs = [{}, {"env_cfg": {"task_name": "finger-spin"}}]
```

- Combining the examples above, here is an example that trains on demos from the current task, random rollouts from the current task, demos from a second task called `"finger-spin"`, and random rollouts from a third task called `"cheetah-run"`:

```
dataset_configs = [
  {},
  {"type": "random"},
  {"env_cfg": {"task_name": "finger-spin"}},
  {"type": "random", "env_cfg": {"task": "cheetah-run"}},
]
```

Since env_cfg_ingredient does not allow for specification of data paths, the configurations passed to auto.load_wds_datasets() also do not allow for paths to be overridden. Instead, the data for a given configuration will always be loaded using the following path template:

```
<data_root>/processed/<data_type>/<task_key>/<benchmark_name>
```

data_root is a config variable from env_data_ingredient, and data_type is the "type" defined in the dataset config dict. "task_key" is env_cfg["task_name"] (which is taken from env_cfg_ingredient by default, but can be overridden in any of the config dicts passed to auto.load_wds_datasets()). Likewise, benchmark_name defaults to env_cfg["benchmark_name"], but can be overridden by dataset config dicts.

3.2.4.2 On-disk format

The webdataset-based on-disk format (which I'll just call the "webdataset format") is very simple: a dataset is composed of 'shards', each of which is a single tar archive. Each tar archive contains a list of files like this:

```
_metadata.meta.pickle
frame_000.acts.pickle
frame_000.dones.pickle
frame_000.frame.pickle
frame_000.infos.pickle
frame_000.next_obs.pickle
frame_000.obs.pickle
frame_000.rews.pickle
frame_001.acts.pickle
frame_001.dones.pickle
frame_001.frame.pickle
frame_001.infos.pickle
frame_001.next_obs.pickle
frame_001.obs.pickle
frame_001.rews.pickle
frame_002.acts.pickle
frame_002.dones.pickle
frame_002.frame.pickle
frame_002.infos.pickle
frame_002.next_obs.pickle
...
```

For the datasets generated by our code, all shards begin with a _metadata.meta.pickle file holding metadata identifying a specific benchmark and task (e.g. it contains the observation space for the task, as well as a configuration for env_data_ingredient that can be used to re-instantiate the whole Gym environment). The remaining files represent time steps in a combined set of trajectories. For instance, the frame_000.* files represent the observation encountered at the first step of the first trajectory, the action taken, the infos dict returned, the next observation encountered, etc. As with the arrays returned by auto.load_dict_dataset(), trajectories are concatenated together in the tar file, and can be separated back out by inspecting the dones values.

Aside: users of the webdataset library usually do not include file-level metadata of the kind stored in _metadata.meta.pickle. Our code has some additional abstractions (such as read_dataset.ILRDataset) which ensure that the file-level metadata is accessible from Python, and which also ensure that _metadata.meta.pickle is not accidentally treated as an additional "frame" when reading the tar file. This is discussed further below.

3.2.4.3 Writing datasets in the webdataset format

Convenience functions for writing datasets are located in data.write_dataset. In particular, this contains a helper function for extracting metadata from an `env_cfg_ingredient` configuration (get_meta_dict()) and a helper for writing a series of frames to an appropriately-structured tar archive (write_frames()). These helpers are currently used by two scripts, which are good resources for understanding how to write webdatasets:

- mkdataset_demos.py: Converts between dict format and webdataset format. That is, the script loads a dataset from its 'native' on-disk format into a dict using `auto.load_dict_dataset()`, then writes the data into a new webdataset.
- mkdataset_random.py: Generates random rollouts on a specified environment and then saves them into a webdataset.

3.2.4.4 Loading data: from shard to minibatch

The main abstraction provided by the webdataset library is the Dataset class. Given a series of URLs pointing to different shards of a dataset, this class iterates over the contents over the shards, one URL at a time. webdataset's `Dataset` is a valid subclass of Torch's `IterableDataset`, so it can be directly passed to Torch's `DataLoader`. A webdataset `Dataset` can also be also be composed with Python generators in order to create a data preprocessing pipeline. For repL, our pipeline looks something like this:

1. **Generic decoding/grouping code:** The first stage of the pipeline does bookkeeping like decoding `.pickle` files in the shard into Python objects (instead of yielding raw bytes as training samples!), and grouping samples with the same frame prefix (e.g. `frame000`, `frame001`, etc.). Our code also uses a special Dataset subclass that makes the contents of `_metadata.meta.pickle` accessible as a dataset instance attribute.
2. **Target pair constructor:** After training samples are decoded, they can be grouped into context and target pairs for the purpose of repL. The TargetPairConstructor interface is simply a generator that processes one sample at a time from the dataset iterator. Since samples are written and read in temporal order, it is possible for these generators to, e.g., create target and context pairs out of temporally adjacent pairs (example).
3. **Optional shuffling:** Since webdataset `Dataset`s are `Iterable` datasets, it is not possible to shuffle the entire dataset in-memory. Instead, the repL code can optionally apply a pipeline stage that buffers a small, fixed number of samples in memory, and pops a randomly-selected sample from this buffer at each iteration. This introduces a small degree of randomisation that may be helpful for optimisation. Note that this step also breaks temporal order, so it must come *after* target pair construction.
4. **Interleaving:** Recall that one of the aims of the `webdataset`-based repL data system was to support multitask training. In principle, we could do this by passing shards from different datasets to webdataset's `Dataset` class. However, since shards are iterated over sequentially (modulo the shuffle buffer), this would mean that the network would exclusively see samples from the first dataset for the first few batches, then exclusively samples from the second dataset, and so on. Instead, we create a separate webdataset `Dataset` for each sub-dataset used for multitask training, and then multiplex those `Dataset`s with InterleavedDataset. `InterleavedDataset` is an `IterableDataset` that repeatedly chooses a sub-dataset uniformly at random and yields a single sample from that. This ensures that the different sub-datasets are equally represented (on average) in each batch.

The steps above yield a single `IterableDataset` which can be passed to Torch's `DataLoader`. The `DataLoader` is then responsible for combining samples from the iterator into batches, just as it would with any other `IterableDataset`.

3.2.5 Adding support for a new benchmark

These are the rough steps required to add support for a new benchmark:

1. Create benchmark-specific routines for creating vec envs; loading data in a dict format; and inferring the equivalent Gym name of an environment. Add these to a module in `il_representations.envs`, much like `il_representations.envs.magical`.
2. Add any required config variables for the new benchmark to `il_representations.envs.config`, and update `il_representations.envs.auto` so that the routines make use of the new config variables to dispatch to the dataset-specific routines in `il_representations.envs.auto`.
3. Update `il_representations.scripts.il_test` to do execute dataset-specific code is required for evaluation of policies in the new environment.
4. Add demonstrations for the new environment to svm and perceptron (in `/scatch/sam/il-demos`). Also update `convert_all_to_new_data_format.sh` (in `il_representations/scripts/`) to produce webdataset-format demonstrations for the new benchmark, and add those to svm/perceptron too. Repeat these steps to copy demonstrations to GCP, too. In particular, if you copy them to `/scratch/sam/il-representations-gcp-volume/il-demos/` in svm or perceptron then they should get automatically synced to GCP.
5. Finally, add configs for one environment from the new benchmark to `test_support.py`, and add test fixtures to `tests/data`. This will make it possible to unit test the new benchmark. Since these data fixtures are stored in the repo, I suggest using only 1-2 trajectories for each fixture.

3.3 Representation Learner Usage

All implementations of representation learning algorithms in this codebase are built around one central abstraction, which is, perhaps not surprisingly, a class called *RepresentationLearner*. This class defines the general framework by which components of representation learner training happen. Different variants of RepL algorithms are designed by creating a learner which takes in and runs different implementations of those component steps.

3.3.1 Training a Pre-Defined Representation Learner

For the sake of our experimentation, we have defined a number of existing, commonly used algorithms, and have made them available to import and use directly.

```
from il_representations.algos import SimCLR, NoAugmentation
from il_representations.utils import convert_to_simple_webdataset, load_simple_webdataset

# This step converts a Pytorch dataset of the form [{'obs': <image>}...] into a Webdataset
# that can stream from disk. This step only needs to be performed once
full_wds_url = convert_to_simple_webdataset(dataset=pytorch_dataset,
                                            file_out_path="temp",
                                            file_out_name="my_dataset")

wds = load_simple_webdataset(full_wds_url)

# For this example, we're imagining a (3, 64, 64) image size
algo = algos.SimCLR(batch_size=10,
                    observation_space=spaces.Box(shape=(3, 64, 64),
                                                 low=0,
```

(continues on next page)

(continued from previous page)

```
                                                high=1),
                    action_space=None,
                    augmenter=NoAugmentation)
# This trains for a single epoch of 10 batches, calculating logging
# information and logging it every step. This is likely substantially
# more logging than you'd want for a typical training use case
algo.learn(datasets=[wds], batches_per_epoch=10, n_epochs=1,
           log_dir='temp', log_interval=1, calc_log_interval=1)
```

3.3.2 Defining a New Representation Learner

Let's use the SimCLR example from above to walk through how you might create an algorithm that differs from it in some way. This is the code used in *algos/__init__.py* to define the SimCLR algorithm class, with some additional explanatory documentation added in for clarity's sake. This explanation will assume you have some familiarity with the conceptual breakdown used in this codebase; if you're unsure about that, you can read more *here!*

```
class SimCLR(RepresentationLearner):
"""
Implementation of SimCLR: A Simple Framework for
Contrastive Learning of Visual Representations
https://arxiv.org/abs/2002.05709

This method works by using a contrastive loss to push together representations
of two differently-augmented versions of the same image. In particular, it
uses a symmetric contrastive loss, which compares the (target, context)
similarity against similarity of context with all other targets, and also
similarity of target with all other contexts.
"""
def __init__(self, **kwargs):
    # This is where we specify the RepresentationLearner arguments
    # that are integral to the algorithm definition of SimCLR

    algo_hardcoded_kwargs = dict(# We use our BaseEncoder to map from image to␣
↪representation
                                 # The output of `encoder` is what we use for transfer
                                 encoder=BaseEncoder,
                                 # A MLP projection head, symmetric between context and␣
↪target
                                 # The output of `decoder` is passed to the loss function
                                 decoder=SymmetricProjectionHead,
                                 # A contrastive loss where we try to predict
                                 # target given context and also context given target
                                 loss_calculator=SymmetricContrastiveLoss,
                                 # Augment both context and target before encoder
                                 augmenter=AugmentContextAndTarget,
                                 # For SimCLR, the target and context are different
                                 # augmentations of the same, "Identity" frame
                                 target_pair_constructor=IdentityPairConstructor,
                                 # Since we're not using momentum here, the encoder
                                 # batch is the same as the one used in the loss
                                 batch_extender=IdentityBatchExtender)
```

(continues on next page)

(continued from previous page)

```
    kwargs = validate_and_update_kwargs(kwargs, algo_hardcoded_kwargs=algo_hardcoded_
↪kwargs)

    super().__init__(**kwargs)
```

3.3.2.1 Existing Pre-Defined Representation Learners

You can find a list of existing algorithms at: `il_representations.algos`

3.4 Design Principles

The design of this repo's core RepresentationLearner abstraction was based around a deconstruction of common RepL learners into their component parts in a way that we feel strikes a good balance between flexibility and reuseability.

To explain a bit more about the different components, let's look at four algorithms: a VAE, a Temporal VAE, SimCLR, and Temporal Contrastive Predictive Coding (CPC). What are the differences between different pairs of these?

- A VAE and TemporalVAE function basically the same way, except that, instead of your reconstruction target being the same as the input frame, it's one frame forward in a trajectory
- Between a VAE and SimCLR, the former tries to reconstruct an input frame after a bottleneck, and the latter tries to achieve similarity with the representation of a differently-augmented input frame after a bottleneck + projection layer. So between these two algorithms, you can identify the differences of (1) using augmentation vs not, and (2) using a contrastive loss rather than a reconstructive one. However, they're the same insofar as the "target" in both cases is (some modification of) the input frame itself. This same analogy holds between TemporalVAE and TemporalCPC: one is reconstructive, one contrastive, but both use a temporally-offset target
- Between SimCLR and TemporalCPC, the central difference is that, instead of calculating a contrastive loss between augmented versions of the same frame, TemporalCPC calculates a contrastive loss between an augmented frame_t and an augmented frame t+k

Now that you've got some practice deconstructing algorithms this way, it may be easier to follow the deconstruction we chose for this codebase. At the most general level, we define representation learners as following the pattern of:

```
L = Loss(Decoder(Encoder(Context), OptionalExtraContext)), Decoder(Encoder(Target))
```

In our framework, different learners are differentiated from one another by their different implementations of each of these components.

1. First, we take in a dataset and construct Context, Target pairs from it. This is done by a *TargetPairConstructor* object. The most common strategies for constructing pairs are identity (where context and target are the same frame) or temporal offset (where context and target are temporally-offset frames). However, there are also situations where the target is not an image input, for example, when we want to predict an action, in the case of inverse dynamics (ID). In that case, *target* is the action vector. *context* objects are always image frames. We also need to handle the case where we need two forms of input information to predict the target: for example, predicting *action* given two contiguous frames in ID, or predicting next frame given current frame and action, in a Dynamics model. These are stored in an optional *extra_context* object, which some encoders have logic to deal with, but which others ignore
2. We augment our context frame (and optionally our target) according to some strategy defined by an *Augmenter*. This is a fairly simple process, and the main variation here is (a) whether to augment both context and target or just context, and (b) what augmentations, if any, to apply.

3. Then, we take our possibly-augmented dataset of Context, Target pairs and run a batch through the *Encoder*. The job of the encoder is to map a context (and optionally also a target) into a z representation vector. This component is what we transfer to downstream models.

4. In some cases, we need to have a *Decoder* to do postprocessing on the representation learned by the encoder, before it is passed to the loss. This component is dropped after RepL training, and not used in downstream finetuning. In the case of a contrastive loss with a projection head, this might be a simple MLP. Or, in the case of a VAE, where loss is calculated on a reconstructed image, this may be a more complex network to reconstruct an image from a bottlenecked representation. Sometimes, it uses *extra_context*, in addition to *context*, to construct an input that can be given to the loss function, as in the cases of Dynamics and Inverse Dynamics mentioned above, where you want to use action vector or next frame respectively as part of the prediction of the other quantity. A decoder may also simply be the identity, in cases where no projection head is used.

5. Once we have run our batches through the encoding and decoding processes, it's time to calculate a loss, with a *RepresentationLoss* object. This loss takes the decoded context, decoded target, and sometimes the encoded context as input. (The latter is basically only used in the case of VAE, where part of our loss is pulling the *p(z|x)* distribution closer to a Gaussian prior).

Given these components, let's compare a few of the definitions of algorithms we gave as examples above.

```
class VariationalAutoencoder(RepresentationLearner):
    """
    A basic variational autoencoder that tries to reconstruct the
    current frame, and calculates a VAE loss over current frame pixels,
    using reconstruction loss and a KL divergence between learned
    z distribution and a normal prior
    """
    def __init__(self, **kwargs):
        # ... <repeated machinery> ...
        algo_hardcoded_kwargs = dict(encoder=VAEEncoder,
                                     decoder=PixelDecoder,
                                     batch_extender=IdentityBatchExtender,
                                     augmenter=NoAugmentation,
                                     loss_calculator=VAELoss,
                                     target_pair_constructor=IdentityPairConstructor,
                                     decoder_kwargs=dict(observation_space=kwargs[
↪'observation_space'],
                                                         encoder_arch_key=dec_encoder_
↪cls_key,
                                                         sample=True))
```

3.5 Interpreting Results

Our current implementation contains some helpful tools for further analyzing the learned policies. The first helps to generate clusters of encoder outputs, and the second analyzes a policy with interpretability algorithms provided in the Captum package.

3.5.1 Generating Clusters

The code to generate clusters is provided in `analysis/clusters.ipynb`. This notebook enables you to get representation encodings from saved policies, and visualize the clusters using Principal component analysis (PCA) or t-distributed Stochastic Neighbor Embedding (t-SNE) in Tensorboard. What it does is to generate and save representations into a `runs` directory, and visualizing it is as simple as running two lines:

```
%load_ext tensorboard
%tensorboard --logdir=runs
```

3.5.2 Interpreting Policies

Sometimes it can be helpful to visualize to what extent a policy relies on certain regions of the state input, and this can be done by algorithms provided by Captum. In `scripts/interpret.py` we incorporate three Primary Attribution methods: Saliency, Integrated Gradients, and DeepLift.

To run the script, you need to specify a few parameters:

- `encoder_path`: The path leading to the saved encoder you want to interpret.
- `chosen_algo` (Optional): The interpretation algorithm you want to run. We currently support one of `['saliency', 'integrated_gradient', 'deep_lift']`, with `'integrated_gradient'` being the default value.
- `length` (Optional): The number of images you want to interpret. This will make the policy interpret the first `length` images from the dataset. The default value is `2`.
- `save_video` (Optional): Whether to save the interpreted `length` images as a video. The default value is `False`.
- `save_image` (Optional): Whether to save `length` images to a local disk. The default value is `True`.
- `device` (Optional): Specify the device you want to run. By default, it will use CUDA if a GPU is available, and use CPU otherwise.

The benchmark and dataset to be tested on by default depends on `env_cfg_defaults` in `envs/config.py`. If you want to specify them on-the-fly, you can set the values when you call this file. Below is an example to run the interpretation on `Procgen`'s `Coinrun` environment:

```
CUDA_VISIBLE_DEVICES=1 python ./src/il_representations/scripts/interpret.py with \
    encoder_path=${path_to_encoder} \
    save_video=True \
    save_image=True \
    chosen_algo=saliency \
    length=1000 \
    env_cfg.benchmark_name=procgen \
    env_cfg.task_name=coinrun
```

3.6 il_representations API documentation

3.6.1 Subpackages

3.6.1.1 il_representations.algos package

Module contents

Submodules

il_representations.algos.augmenters module

il_representations.algos.batch_extenders module

il_representations.algos.decoders module

il_representations.algos.encoders module

il_representations.algos.losses module

il_representations.algos.optimizers module

il_representations.algos.pair_constructors module

il_representations.algos.representation_learner module

il_representations.algos.utils module

3.6.1.2 il_representations.configs package

Module contents

This package contains Sacred configs for our experiment scripts.

Submodules

il_representations.configs.chain_configs module

il_representations.configs.experimental_conditions module

il_representations.configs.hp_tuning module

il_representations.configs.icml_experiment_configs module

il_representations.configs.icml_hp_tuning module

il_representations.configs.joint_training_configs module

il_representations.configs.run_rep_learner_configs module

3.6.1.3 il_representations.data package

Module contents

This package contains utilities for reading and writing datasets in our webdataset-based format.

Submodules

il_representations.data.read_dataset module

il_representations.data.write_dataset module

3.6.1.4 il_representations.envs package

Module contents

The *envs* package contains code for instantiating environments, representing environment metadata, etc.

Submodules

il_representations.envs.atari_envs module

il_representations.envs.auto module

il_representations.envs.baselines_vendored module

il_representations.envs.config module

il_representations.envs.dm_control_envs module

il_representations.envs.magical_envs module

il_representations.envs.minecraft_envs module

il_representations.envs.procgen_envs module

il_representations.envs.utils module

3.6.1.5 il_representations.il package

Module contents

The *il* package contains re-implementations of IL algorithms used in our joint training experiments.

Submodules

il_representations.il.bc module

il_representations.il.bc_support module

il_representations.il.disc_rew_nets module

il_representations.il.gail_pol_save module

class il_representations.il.gail_pol_save.**GAILSavePolicyCallback**(*ppo_algo*, *save_every_n_steps*, *save_dir*, *, *save_template='policy_{timesteps:08d}_steps.pt'*)

Bases: object

This callback can be passed to AdversarialTrainer.train() to save a policy snapshot every *save_every_n_steps* time steps.

il_representations.il.score_logging module

SB3 score-logging callback for MAGICAL (but should be safe to add to include when using any environment—if the desired *eval_score* key is not in *infos*, then it won't add any log entries).

class il_representations.il.score_logging.**SB3ScoreLoggingCallback**(**args: Any*, ***kwargs: Any*)

Bases: stable_baselines3.common.callbacks.stable_baselines3.common.callbacks.BaseCallback._name

Callback for SB3 RL algorithms which extracts the 'eval_score' from the step info dict (if it exists) and includes it in the logs. Useful for MAGICAL, which reports end-of-trajectory performance using *eval_score*.

Tested for PPO, but may work for other algorithms too.

il_representations.il.utils module

Utilities that are helpful for several pieces of IL code (e.g. in both il_train.py and joint_training.py).

il_representations.il.utils.**add_infos**(*data_iter*)

Add a dummy 'infos' value to each dict in a data stream.

il_representations.il.utils.**streaming_extract_keys**(**keys_to_keep*)

Filter a generator of dicts to keep only the specified keys.

3.6.1.6 il_representations.scripts package

Module contents

The *scripts* package contains the scripts that we use to perform representation learning, imitation learning, evaluation, dataset generation, and so on.

Subpackages

il_representations.scripts.data package

Module contents

The *scripts.data* package contains scripts for working with datasets.

Submodules

il_representations.scripts.collate_configs module

il_representations.scripts.grab_some_stuff_and_pickle_it module

il_representations.scripts.il_test module

il_representations.scripts.il_train module

il_representations.scripts.interpret module

il_representations.scripts.joint_training module

il_representations.scripts.joint_training_cluster module

il_representations.scripts.mkdataset_demos module

il_representations.scripts.mkdataset_random module

il_representations.scripts.pretrain_n_adapt module

il_representations.scripts.render_dataset module

il_representations.scripts.run_rep_learner module

il_representations.scripts.save_traced_net module

Trace & save a network for a MAGICAL environment. Capable of automatically figuring out where the encoder is.

il_representations.scripts.save_traced_net.**auto_save_name**(*module_path: str*) → str
Use config.json files to automatically come up with a name for the given encoder.

il_representations.scripts.save_traced_net.**fetch_encoder**(*net: torch.nn.Module*) → torch.nn.Module

il_representations.scripts.save_traced_net.**get_config_path**(*module_path: str*) → Optional[str]
Keep walking up the tree until we find a config.json.

il_representations.scripts.save_traced_net.**load_eval_net**(*net_path: str*) → torch.nn.Module
Load a network on disk, making sure it ends up on the CPU & in eval mode.

il_representations.scripts.save_traced_net.**main**(*args: argparse.Namespace*) → None

il_representations.scripts.save_traced_net.**pre_order_children**(*net: torch.nn.Module*) → Iterator[Tuple[str, torch.nn.Module]]

Traverse module tree in pre-order.

il_representations.scripts.save_traced_net.**trace_encoder**(*encoder: torch.nn.Module*) → torch.nn.Module

Generate a random example of the appropriate size & use it to trace the given network.

il_representations.scripts.truncate_datasets_icml module

3.6.1.7 il_representations.test_support package

Module contents

The *test_support* package contains code used only in unit tests.

Submodules

il_representations.test_support.configuration module

il_representations.test_support.utils module

3.6.2 Module contents

3.6.3 Submodules

3.6.4 il_representations.pol_eval module

3.6.5 il_representations.policy_interfacing module

3.6.6 il_representations.script_utils module

3.6.7 il_representations.utils module

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX