VISSL Documentation

Release 0.1.5

vissl contributors

Mar 02, 2021

INDEX

VISSL is a computer vision library for state-of-the-art Self-Supervised Learning research with PyTorch. VISSL aims to accelerate research cycle in self-supervised learning: from designing a new self-supervised task to evaluating the learned representations.

CHAPTER

ONE

WHAT IS VISSL?

VISSL is a computer VIsion library for state-of-the-art Self-Supervised Learning research with PyTorch. VISSL aims to accelerate research cycle in self-supervised learning: from designing a new self-supervised task to evaluating the learned representations. Key features include:

- `Reproducible implementation of SOTA in Self-Supervision`: All existing SOTA in Self-Supervision are implemented - SwAV, SimCLR, MoCo(v2), PIRL, NPID, NPID++, DeepClusterV2, ClusterFit, RotNet, Jigsaw. Also supports supervised trainings.
- `Benchmark suite`: Variety of benchmarks tasks including linear image classification (places205, imagenet1k, voc07, inaturalist), full finetuning, semi-supervised benchmark, nearest neighbor benchmark, object detection (Pascal VOC and COCO).
- `Ease of Usability`: easy to use using yaml configuration system based on Hydra.
- `Modular`: Easy to design new tasks and reuse the existing components from other tasks (objective functions, model trunk and heads, data transforms, etc.). The modular components are simple *drop-in replacements* in yaml config files.
- `Scalability`: Easy to train model on 1-gpu, multi-gpu and multi-node. Several components for large scale trainings provided as simple config file plugs: Activation checkpointing, ZeRO, FP16, LARC, Stateful data sampler, data class to handle invalid images, large model backbones like RegNets, etc.
- `Model Zoo`: Over *60 pre-trained self-supervised model* weights.

We hope that VISSL will democratize self-supervised learning and accelerate advancements in self-supervised learning. We also hope that it will enable research in some important research directions like Generalizability of models

etc.

Hope you enjoy using VISSL!

CHAPTER

TWO

INSTALLATION

Our installation is simple and we provide pre-built binaries (pip, conda) and also instructions for building from source (pip, conda).

2.1 Requirements

At a high level, project requires following system dependencies.

- Linux
- Python>=3.6.2 and <3.9
- PyTorch>=1.4
- torchvision (matching PyTorch install)
- CUDA (must be a version supported by the pytorch version)
- OpenCV (Optional)

2.2 Installing VISSL from pre-built binaries

2.2.1 Install VISSL conda package

This assumes you have conda 10.2.

```
conda create -n vissl python=3.8
conda activate vissl
conda install -c pytorch pytorch=1.7.1 torchvision cudatoolkit=10.2
conda install -c vissl -c iopath -c conda-forge -c pytorch -c defaults apex vissl
```

For other versions of PyTorch, Python, CUDA, please modify the above instructions with the desired version. VISSL provides Apex packages for all combinations of pytorch, python and compatible cuda.

2.2.2 Install VISSL pip package

This example is with pytorch 1.5.1 and cuda 10.1. Please modify the PyTorch version, cuda version and accordingly apex version below for the desired settings.

- **Step 1: Create Virtual environment (pip)**

```
python3 -m venv ~/venv
. ~/venv/bin/activate
```

- **Step 2: Install PyTorch, OpenCV and APEX (pip)**
  - We use PyTorch=1.5.1 with CUDA 10.1 in the following instruction (user can chose their desired version).
  - There are several ways to install opencv, one possibility is as follows.
  - For APEX, we provide pre-built binary built with optimized C++/CUDA extensions provided by APEX.

```
pip install torch==1.5.1+cu101 torchvision==0.6.1+cu101 -f https://download.pytorch.
↪org/whl/torch_stable.html
pip install opencv-python
pip install apex -f https://dl.fbaipublicfiles.com/vissl/packaging/apexwheels/py38_
↪cu101_pyt151/download.html
```

Note that, for the APEX install, you need to get the versions of CUDA, PyTorch, and Python correct in the URL. We provide APEX versions with all possible combinations of Python, PyTorch, CUDA. Select the right APEX Wheels if you desire a different combination.

On Google Colab, everything until this point is already set up. You install APEX there as follows.

```
import sys
import torch
version_str="".join([
    f"py3{sys.version_info.minor}_cu",
    torch.version.cuda.replace(".",""),
    f"_pyt{torch.__version__[0:5:2]}"
])
!pip install apex -f https://dl.fbaipublicfiles.com/vissl/packaging/apexwheels/
↪{version_str}/download.html
```

- **Step 3: Install VISSL**

```
pip install vissl
# verify installation
python -c 'import vissl'
```

2.3 Installing VISSL from source

The following instructions assume that you have desired CUDA version installed and working.

2.3.1 Install from source in PIP environment

- **Step 1: Create Virtual environment (pip)**

```
python3 -m venv ~/venv
. ~/venv/bin/activate
```

- **Step 2: Install PyTorch (pip)**

```
pip install torch==1.7.1+cu101 torchvision==0.8.2+cu101 -f https://download.pytorch.
↪org/whl/torch_stable.html
```

- **Step 3: Install APEX (pip)**

```
pip install apex -f https://dl.fbaipublicfiles.com/vissl/packaging/apexwheels/py37_
↪cu101_pyt171/download.html
```

- **Step 4: Install VISSL**

```
# clone vissl repository
cd $HOME && git clone --recursive https://github.com/facebookresearch/vissl.git && cd
↪$HOME/vissl/
# install vissl dependencies
pip install --progress-bar off -r requirements.txt
pip install opencv-python
# update classy vision install to current master
pip uninstall -y classy_vision
pip install classy-vision@https://github.com/facebookresearch/ClassyVision/tarball/
↪master
# install vissl dev mode (e stands for editable)
pip install -e .[dev]
# verify installation
python -c 'import vissl, apex, cv2'
```

2.3.2 Install from source in Conda environment

- **Step 1: Create Conda environment**

If you don't have anaconda, run this bash scrip to install conda.

```
conda create -n vissl_env python=3.7
source activate vissl_env
```

- **Step 2: Install PyTorch (conda)**

```
conda install pytorch torchvision cudatoolkit=10.1 -c pytorch
```

- **Step 3: Install APEX (conda)**

```
conda install -c vissl apex
```

- **Step 4: Install VISSL**

Follow step4 instructions from the PIP installation above.

That's it! You are now ready to use this code.

- **Optional: Install Apex from source (common for both pip and conda)**

Apex installation requires that you have a latest nvcc so the c++ extensions can be compiled with latest gcc (>=7.4). Check the APEX website for more instructions.

```
# see https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-
↪architecture-feature-list
# to select cuda architecture you want to build
CUDA_VER=10.1 TORCH_CUDA_ARCH_LIST="5.0;5.2;5.3;6.0;6.1;6.2;7.0;7.5" ./docker/common/
↪install_apex.sh
```

CHAPTER

THREE

GETTING STARTED WITH VISSL

This document provides a brief introduction of usage of built-in command line tools provided by VISSL.

3.1 Quick Start with VISSL

We provide a quick overview for training SimCLR self-supervised model on 1-gpu with VISSL.

3.2 Install VISSL

For installation, please follow our installation instructions.

3.3 Setup dataset

We will use ImageNet-1K dataset and assume the downloaded data to look like:

```
imagenet_full_size
    |_ train
    |  |_ <n0......>
    |  |  |_<im-1-name>.JPEG
    |  |  |_...
    |  |  |_<im-N-name>.JPEG
    |  |_ ...
    |  |_ <n1......>
    |  |  |_<im-1-name>.JPEG
    |  |  |_...
    |  |  |_<im-M-name>.JPEG
    |  |  |_...
    |  |  |_...
    |_ val
    |  |_ <n0......>
    |  |  |_<im-1-name>.JPEG
    |  |  |_...
    |  |  |_<im-N-name>.JPEG
    |  |_ ...
    |  |_ <n1......>
    |  |  |_<im-1-name>.JPEG
    |  |  |_...
    |  |  |_<im-M-name>.JPEG
```

(continues on next page)

(continued from previous page)

```
    |  |  |_...
    |  |  |_...
```

3.4 Running SimCLR Pre-training on 1-gpu

3.4.1 If VISSL is built from source

We provide a config to train model using the pretext SimCLR task on the ResNet50 model. Change the `DATA.TRAIN.DATA_PATHS` path to the ImageNet train dataset folder path.

```
python3 run_distributed_engines.py \
    hydra.verbose=true \
    config.DATA.TRAIN.DATASET_NAMES=[imagenet1k_folder] \
    config.DATA.TRAIN.DATA_SOURCES=[disk_folder] \
    config.DATA.TRAIN.DATA_PATHS=["/path/to/my/imagenet/folder/train"] \
    config=test/integration_test/quick_simclr \
    config.CHECKPOINT.DIR="./checkpoints" \
    config.TENSORBOARD_SETUP.USE_TENSORBOARD=true
```

3.4.2 If using pre-built conda/pip VISSL packages

Users need to set the dataset and obtain the builtin tool for training. Follow the steps:

- **Step1: Setup ImageNet1K dataset**

If you installed pre-built VISSL packages, we will set the ImageNet1K dataset following our data documentation and tutorial. NOTE that we need to register the dataset with VISSL.

In your python interpretor:

```
>>> json_data = {
        "imagenet1k_folder": {
            "train": ["<img_path>", "<lbl_path>"],
            "val": ["<img_path>", "<lbl_path>"]
        }
    }
>>> from vissl.utils.io import save_file
>>> save_file(json_data, "/tmp/configs/config/dataset_catalog.json")
>>> from vissl.data.dataset_catalog import VisslDatasetCatalog
>>> print(VisslDatasetCatalog.list())
['imagenet1k_folder']
>>> print(VisslDatasetCatalog.get("imagenet1k_folder"))
{'train': ['<img_path>', '<lbl_path>'], 'val': ['<img_path>', '<lbl_path>']}
```

- **Step2: Get the builtin tool and yaml config file**

We will use the pre-built VISSL tool for training run_distributed_engines.py and the config file. Run

```
cd /tmp/ && mkdir -p /tmp/configs/config
wget -q -O configs/__init__.py https://dl.fbaipublicfiles.com/vissl/tutorials/configs/
↪__init__.py
wget -q -O configs/config/quick_1gpu_resnet50_simclr.yaml https://dl.fbaipublicfiles.
↪com/vissl/tutorials/configs/quick_1gpu_resnet50_simclr.yaml
wget -q  https://dl.fbaipublicfiles.com/vissl/tutorials/run_distributed_engines.py
```

- **Step3: Train**

```
cd /tmp/
python3 run_distributed_engines.py \
    hydra.verbose=true \
    config.DATA.TRAIN.DATASET_NAMES=[imagenet1k_folder] \
    config.DATA.TRAIN.DATA_SOURCES=[disk_folder] \
    config.DATA.TRAIN.DATA_PATHS=["/path/to/my/imagenet/folder/train"] \
    config=quick_1gpu_resnet50_simclr \
    config.CHECKPOINT.DIR="./checkpoints" \
    config.TENSORBOARD_SETUP.USE_TENSORBOARD=true
```

CHAPTER

FOUR

YAML CONFIGURATION SYSTEM

VISSL uses Hydra for configuration management. The configuration files are simple YAML files. Hydra provides flexible yet powerful configuration system.

- Users can create configs for only a specific component of their training (for example: using different datasets) and overwrite a master configuration setting for that specific component. This way, Hydra allows reusability of configs.
- Hydra also allows to modify the configuration values from command line and
- Hydra also offers an intuitive solution to adding new keys to a configuration.

The usage looks like:

```
python <binary-name>.py config=<yaml_config path>/<yaml_config_file_name>
```

4.1 Detecting new configuration directories in Hydra

VISSL provides configuration files here and uses the Hydra Plugin VisslPlugin to automatically search for the `configs` folder in VISSL.

If users want to create their own configuration directories and not use the `configs` directory provided by VISSL, then users must add their own Plugin following the `VisslPlugin`.

Note: For any new folder containing configuration files, Hydra requires creating a `__init__.py` empty file. Hence, if users create a new configuration directory, they must create empty `__init__.py` file.

4.2 How to use VISSL provided config files

For example, to train SwAV model on 8-nodes (32-gpu) with VISSL:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet
```

where `swav_8node_resnet.yaml` is a master configuration file for SwAV training and exists at `vissl/configs/config/pretrain/swav/swav_8node_resnet.yaml`.

4.3 How to add configuration files for new SSL approaches

Let's say you have a new self-supervision approach that you implemented in VISSL and want to create config files for training. You can simply create a new folder and config file for your approach.

For example:

```
python tools/run_distributed_engines.py \
  config=pretrain/my_new_approach/my_approach_config_file.yaml
```

In the above case, we are simply creating the `my_new_approach` folder under `pretrain/` path and create a file `my_approach_config_file.yaml` with the path *pretrain/my_new_approach/my_approach_config_file.yaml*

4.4 How to override a training component with config files

To replace one training component with the other, for example, replacing the training datasets, one can achieve this by simply creating a new yaml file for the dataset and use that during training.

For example:

```
python tools/run_distributed_engines.py \
  config=pretrain/swav/swav_8node_resnet \
  +config/pretrain/swav/optimization=my_new_optimization \
  +config/pretrain/swav/my_new_dataset=my_new_dataset_file_name \
```

In the above case, we are overriding optimization and data settings for the SwAV training. For overriding, we simply create the `my_new_dataset` sub-folder under `pretrain/swav` path and create a file `my_new_dataset_file_name.yaml` with the path *pretrain/swav/my_new_dataset_file_name.yaml*

4.5 How to override single values in config files

If you want to override single value of an existing key in the config, you can achieve that with: `my_key=my_new_value`

For example:

```
python tools/run_distributed_engines.py \
    config=pretrain/swav/swav_8node_resnet \
    config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights_path.torch>
```

4.6 How to add new keys to the dictionary in config files

If you want to add single key to a dictionary in the config, you can achieve that with `+my_new_key_name=my_value`. Note the use of `+`.

For example:

```
python tools/run_distributed_engines.py \
    config=pretrain/swav/swav_8node_resnet \
    +config.MY_NEW_KEY=MY_VALUE \
    +config.LOSS.simclr_info_nce_loss.MY_NEW_KEY=MY_VALUE
```

CHAPTER

FIVE

USING TENSORBOARD IN VISSL

VISSL provides integration of Tensorboard to facilitate self-supervised training and experimentation. VISSL logs many useful scalars and non-scalars to Tensorboard that provide useful insights into an ongoing training.:

- **Scalars**:
  - Training Loss
  - Learning Rate
  - Average Training iteration time
  - Batch size per gpu
  - Number of images per sec per gpu
  - Training ETA
  - GPU memory used
  - Peak GPU memory allocated
- **Non-scalars**:
  - Model parameters (at the start of every epoch and/or after N iterations)
  - Model parameter gradients (at the start of every epoch and/or after N iterations)

5.1 How to use Tensorboard in VISSL

Using Tensorboard is very easy in VISSL and can be achieved by setting some configuration options. User needs to set `TENSORBOARD_SETUP.USE_TENSORBOARD=true` and adjust the values of other config parameters as desired. Full set of parameters exposed by VISSL for Tensorboard:

```
HOOKS:
  TENSORBOARD_SETUP:
    # whether to use tensorboard for the visualization
    USE_TENSORBOARD: False
    # log directory for tensorboard events
    LOG_DIR: "."
    EXPERIMENT_LOG_DIR: "tensorboard"
    # flush logs every n minutes
    FLUSH_EVERY_N_MIN: 5
    # whether to log the model parameters to tensorboard
    LOG_PARAMS: True
    # whether ttp log the model parameters gradients to tensorboard
    LOG_PARAMS_GRADIENTS: True
```

(continues on next page)

(continued from previous page)

```
    # if we want to log the model parameters every few iterations, set the iteration
    # frequency. -1 means the params will be logged only at the end of epochs.
    LOG_PARAMS_EVERY_N_ITERS: 310
```

Note: Please install tensorboard manually: if pip environment: `pip install tensorboard` or if using conda and you prefer conda install of tensorboard: `conda install -c conda-forge tensorboard`.

5.2 Example usage

For example, to use Tensorboard during SwAV training, the command would look like:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.TENSORBOARD_SETUP.USE_TENSORBOARD=true \
    config.TENSORBOARD_SETUP.LOG_PARAMS=true \
    config.TENSORBOARD_SETUP.LOG_PARAMS_GRADIENTS=true \
    config.TENSORBOARD_SETUP.LOG_DIR=/tmp/swav_tensorboard_events/
```

CHAPTER

SIX

COMPATIBILITY WITH OTHER LIBRARIES

- VISSL provides several helpful scripts to convert VISSL models to models that are compatible with other libraries like Detectron2 and ClassyVision compatible models.
- VISSL also provides scripts to convert models from other sources like Caffe2 models in the paper to VISSL compatible models.
- TorchVision models are directly compatible with VISSL and don't require any conversion.

6.1 Converting Models VISSL -> {Detectron2, ClassyVision, TorchVision}

We provide scripts to convert VISSL models to Detectron2 and ClassyVision compatible models.

6.1.1 Converting to Detectron2

All the ResNe(X)t models in VISSL can be converted to Detectron2 weights using following command:

```
python extra_scripts/convert_vissl_to_detectron2.py \
    --input_model_file <input_model>.pth  \
    --output_model <d2_model>.torch \
    --weights_type torch \
    --state_dict_key_name classy_state_dict
```

6.1.2 Converting to ClassyVision

All the ResNe(X)t models in VISSL can be converted to Detectron2 weights using following command:

```
python extra_scripts/convert_vissl_to_classy_vision.py \
    --input_model_file <input_model>.pth  \
    --output_model <d2_model>.torch \
    --state_dict_key_name classy_state_dict
```

6.1.3 Converting to TorchVision

All the ResNe(X)t models in VISSL can be converted to Torchvision weights using following command:

```
python extra_scripts/convert_vissl_to_torchvision.py \
    --model_url_or_file <input_model>.pth  \
    --output_dir /path/to/output/dir/ \
    --output_name <my_converted_model>.torch
```

6.2 Converting Caffe2 models -> VISSL

We provide conversion of all the Caffe2 models in the paper.

6.2.1 ResNet-50 models to VISSL

- **Jigsaw model:**

```
python extra_scripts/convert_caffe2_to_torchvision_resnet.py \
    --c2_model <model>.pkl \
    --output_model <pth_model>.torch \
    --jigsaw True --bgr2rgb True
```

- **Colorization model:**

```
python extra_scripts/convert_caffe2_to_torchvision_resnet.py \
    --c2_model <model>.pkl \
    --output_model <pth_model>.torch \
    --bgr2rgb False
```

- **Supervised model:**

```
python extra_scripts/convert_caffe2_to_pytorch_rn50.py \
    --c2_model <model>.pkl \
    --output_model <pth_model>.torch \
    --bgr2rgb True
```

6.2.2 AlexNet models to VISSL

- **AlexNet Jigsaw models:**

```
python extra_scripts/convert_caffe2_to_vissl_alexnet.py \
    --weights_type caffe2 \
    --model_name jigsaw \
    --bgr2rgb True \
    --input_model_weights <model.pkl> \
    --output_model <pth_model>.torch
```

- **AlexNet Colorization models:**

```
python extra_scripts/convert_caffe2_to_vissl_alexnet.py \
    --weights_type caffe2 \
    --model_name colorization \
```

(continues on next page)

(continued from previous page)

```
    --input_model_weights <model.pkl> \
    --output_model <pth_model>.torch
```

- **AlexNet Supervised models:**

```
python extra_scripts/convert_caffe2_to_vissl_alexnet.py \
    --weights_type caffe2 \
    --model_name supervised \
    --bgr2rgb True \
    --input_model_weights <model.pkl> \
    --output_model <pth_model>.torch
```

6.3 Converting Models ClassyVision -> VISSL

We provide scripts to convert ClassyVision models to VISSL compatible models.

```
python extra_scripts/convert_classy_vision_to_vissl_resnet.py \
    --input_model_file <input_model>.pth  \
    --output_model <d2_model>.torch \
    --depth 50
```

6.4 Converting Official RotNet and DeepCluster models -> VISSL

- **AlexNet RotNet model:**

```
python extra_scripts/convert_caffe2_to_vissl_alexnet.py \
    --weights_type torch \
    --model_name rotnet \
    --input_model_weights <model> \
    --output_model <pth_model>.torch
```

- **AlexNet DeepCluster model:**

```
python extra_scripts/convert_alexnet_models.py \
    --weights_type torch \
    --model_name deepcluster \
    --input_model_weights <model> \
    --output_model <pth_model>.torch
```

CHAPTER

SEVEN

CONTRIBUTING TO VISSL

We want to make contributing to this project as easy and transparent as possible.

7.1 Our Development Process

Minor changes and improvements will be released on an ongoing basis. Larger changes (e.g., changesets implementing a new SSL approach, benchmark, new scaling feature etc) will be released on a more periodic basis.

7.2 Issues

We use GitHub issues to track public bugs and questions. Please make sure to follow one of the issue templates when reporting any issues.

Facebook has a bounty program for the safe disclosure of security bugs. In those cases, please go through the process outlined on that page and do not file a public issue.

7.3 Pull Requests

We actively welcome your pull requests.

However, if you're adding any significant features (e.g. > 50 lines), please make sure to have a corresponding issue to discuss your motivation and proposals, before sending a PR. We do not always accept new features, and we take the following factors into consideration:

1. Whether the same feature can be achieved without modifying VISSL. VISSL is designed to be extensible so that it's easy to extend any modular component and train custom models. If some part is not as extensible, you can also bring up the issue to make it more extensible.
2. Whether the feature is potentially useful to a large audience, or only to a small portion of users.
3. Whether the proposed solution has a good design / interface.
4. Whether the proposed solution adds extra mental/practical overhead to users who don't need such feature.
5. Whether the proposed solution breaks existing APIs.

When sending a PR, please do:

1. Fork the repo and create your branch from `master`.
2. If a PR contains multiple orthogonal changes, split it to several PRs.

3. If you've added code that should be tested, add tests.
4. If you've changed APIs, update the documentation.
5. Ensure the test suite passes. Follow cpu test instructions and integration tests.
6. Make sure your code follows our coding practices (see next section).
7. If you haven't already, complete the Contributor License Agreement ("CLA").

7.4 Coding Style

Please follow our coding practices and choose either option to properly format your code before submitting PRs.

7.5 Contributor License Agreement ("CLA")

In order to accept your pull request, we need you to submit a CLA. You only need to do this once to work on any of Facebook's open source projects.

Complete your CLA here.

7.6 License

By contributing to ssl_framework, you agree that your contributions will be licensed under the LICENSE file in the root directory of this source tree.

CHAPTER

EIGHT

CONTACT

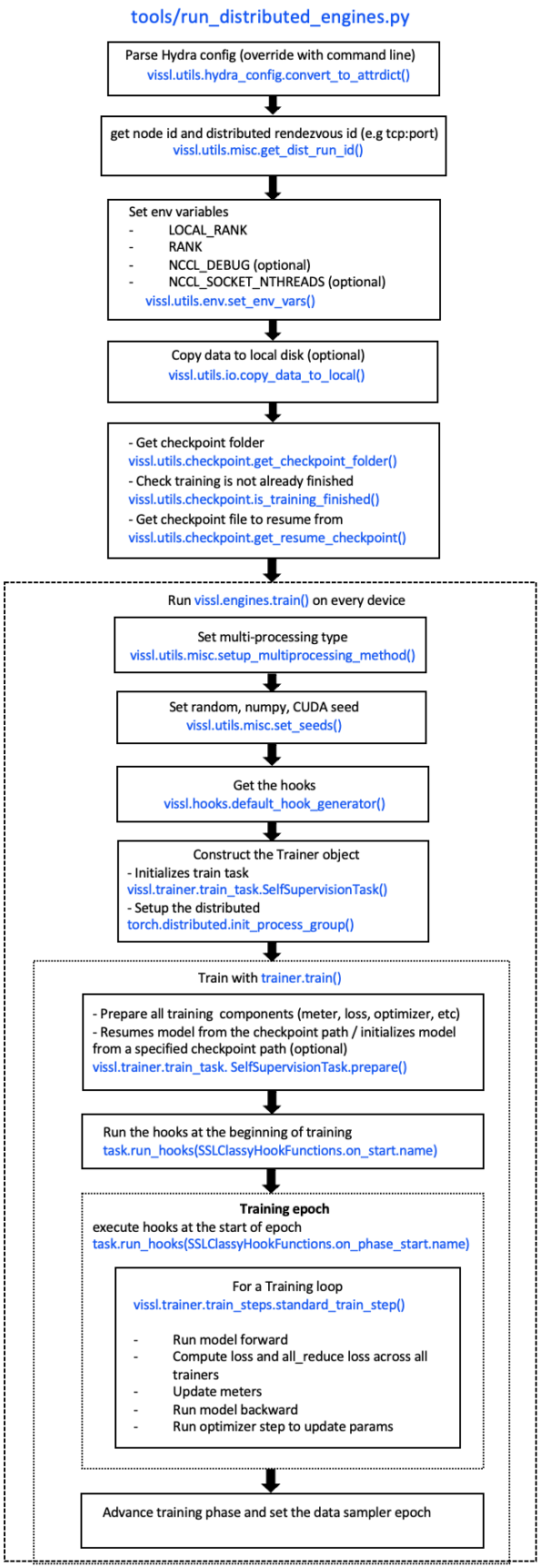
If you want to contact the team about something else than code, like an idea for collaboration, drop us an email at vissl@fb.com.

CHAPTER

NINE

TRAINING: STEP-BY-STEP EXECUTION

We demonstrate step-by-step execution of how training works in VISSL in the following flowchart.

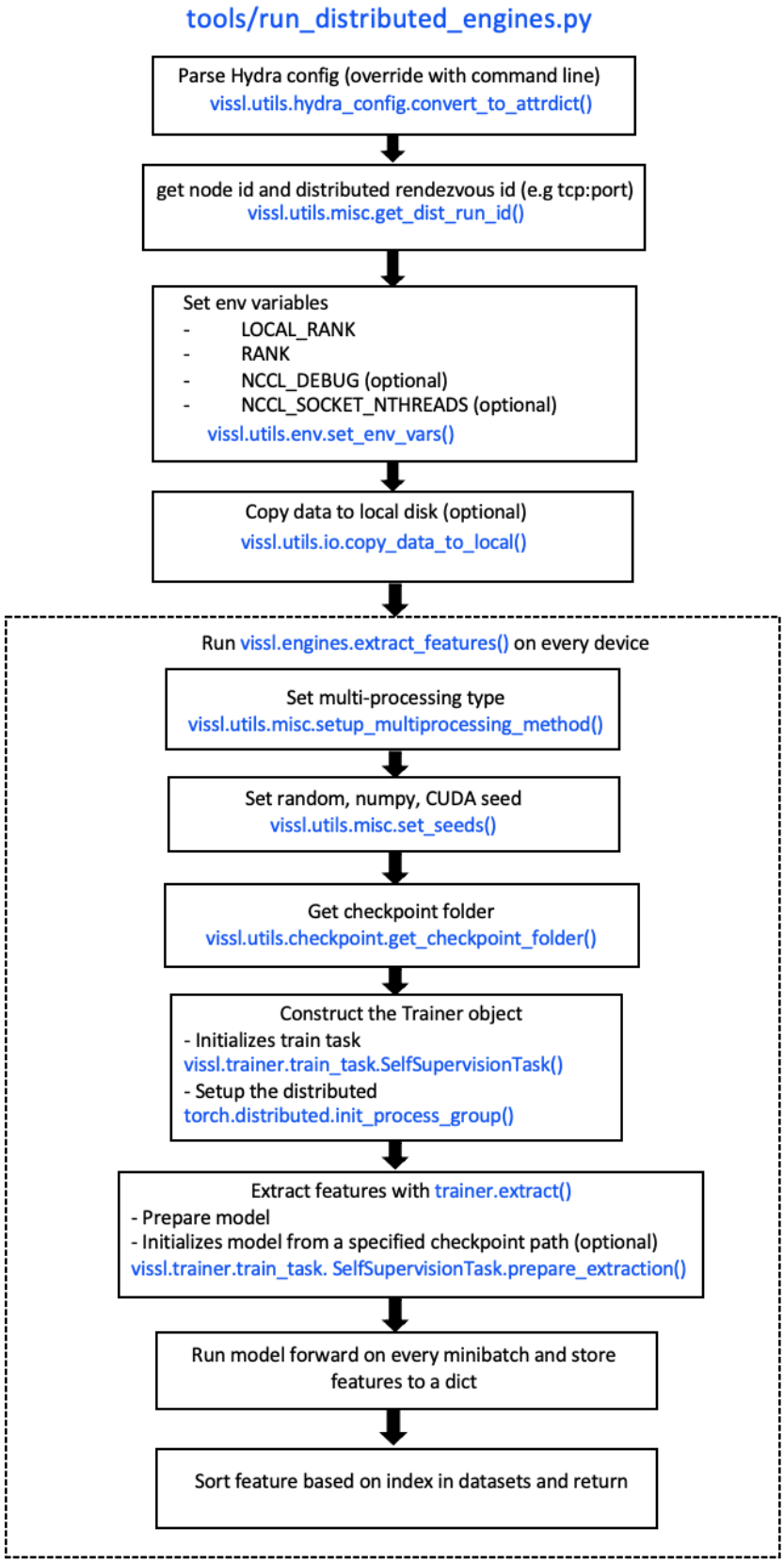

tools/run_distributed_engines.py
Parse Hydra config (override with command line)
vissl.utils.hydra_config.convert_to_attrdict()
get node id and distributed rendezvous id (e.g tcp:port)
vissl.utils.misc.get_dist_run_id()
Set env variables
- LOCAL_RANK
- RANK
- NCCL_DEBUG (optional)
- NCCL_SOCKET_NTHREADS (optional)
vissl.utils.env.set_env_vars()
Copy data to local disk (optional)
vissl.utils.io.copy_data_to_local()
- Get checkpoint folder
vissl.utils.checkpoint.get_checkpoint_folder()
- Check training is not already finished
vissl.utils.checkpoint.is_training_finished()
- Get checkpoint file to resume from
vissl.utils.checkpoint.get_resume_checkpoint()
Run vissl.engines.train() on every device
Set multi-processing type
vissl.utils.misc.setup_multiprocessing_method()
Set random, numpy, CUDA seed
vissl.utils.misc.set_seeds()
Get the hooks
vissl.hooks.default_hook_generator()
Construct the Trainer object
- Initializes train task
vissl.trainer.train_task.SelfSupervisionTask()
- Setup the distributed
torch.distributed.init_process_group()
Train with trainer.train()
- Prepare all training components (meter, loss, optimizer, etc)
- Resumes model from the checkpoint path / initializes model from a specified checkpoint path (optional)
vissl.trainer.train_task. SelfSupervisionTask.prepare()
Run the hooks at the beginning of training
task.run_hooks(SSLClassyHookFunctions.on_start.name)
Training epoch
execute hooks at the start of epoch
task.run_hooks(SSLClassyHookFunctions.on_phase_start.name)
For a Training loop
vissl.trainer.train_steps.standard_train_step()
- Run model forward
- Compute loss and all_reduce loss across all trainers
- Update meters
- Run model backward
- Run optimizer step to update params
Advance training phase and set the data sampler epoch

CHAPTER

TEN

FEATURE EXTRACTION: STEP-BY-STEP EXECUTION

We demonstrate step-by-step how feature extraction happens in VISSL in the following flowchart.

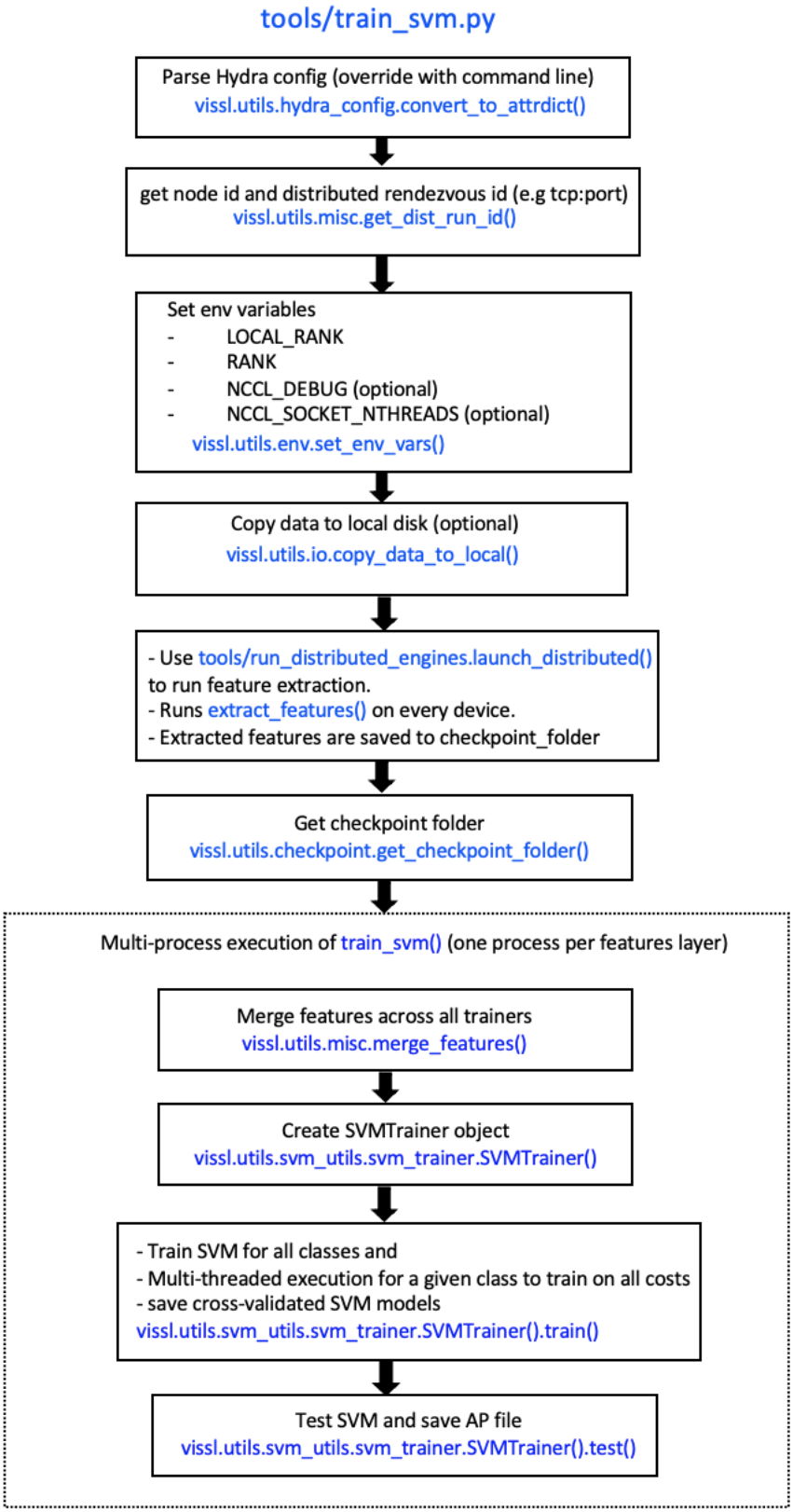

tools/run_distributed_engines.py
Parse Hydra config (override with command line)
vissl.utils.hydra_config.convert_to_attrdict()
get node id and distributed rendezvous id (e.g tcp:port)
vissl.utils.misc.get_dist_run_id()
Set env variables
- LOCAL_RANK
- RANK
- NCCL_DEBUG (optional)
- NCCL_SOCKET_NTHREADS (optional)
vissl.utils.env.set_env_vars()
Copy data to local disk (optional)
vissl.utils.io.copy_data_to_local()
Run vissl.engines.extract_features() on every device
Set multi-processing type
vissl.utils.misc.setup_multiprocessing_method()
Set random, numpy, CUDA seed
vissl.utils.misc.set_seeds()
Get checkpoint folder
vissl.utils.checkpoint.get_checkpoint_folder()
Construct the Trainer object
- Initializes train task
vissl.trainer.train_task.SelfSupervisionTask()
- Setup the distributed
torch.distributed.init_process_group()
Extract features with trainer.extract()
- Prepare model
- Initializes model from a specified checkpoint path (optional)
vissl.trainer.train_task. SelfSupervisionTask.prepare_extraction()
Run model forward on every minibatch and store features to a dict
Sort feature based on index in datasets and return

CHAPTER

ELEVEN

BENCHMARK ON VOC07: STEP-BY-STEP EXECUTION

We demonstrate step-by-step execution of SVM training benchmark in VISSL in the following flowchart.

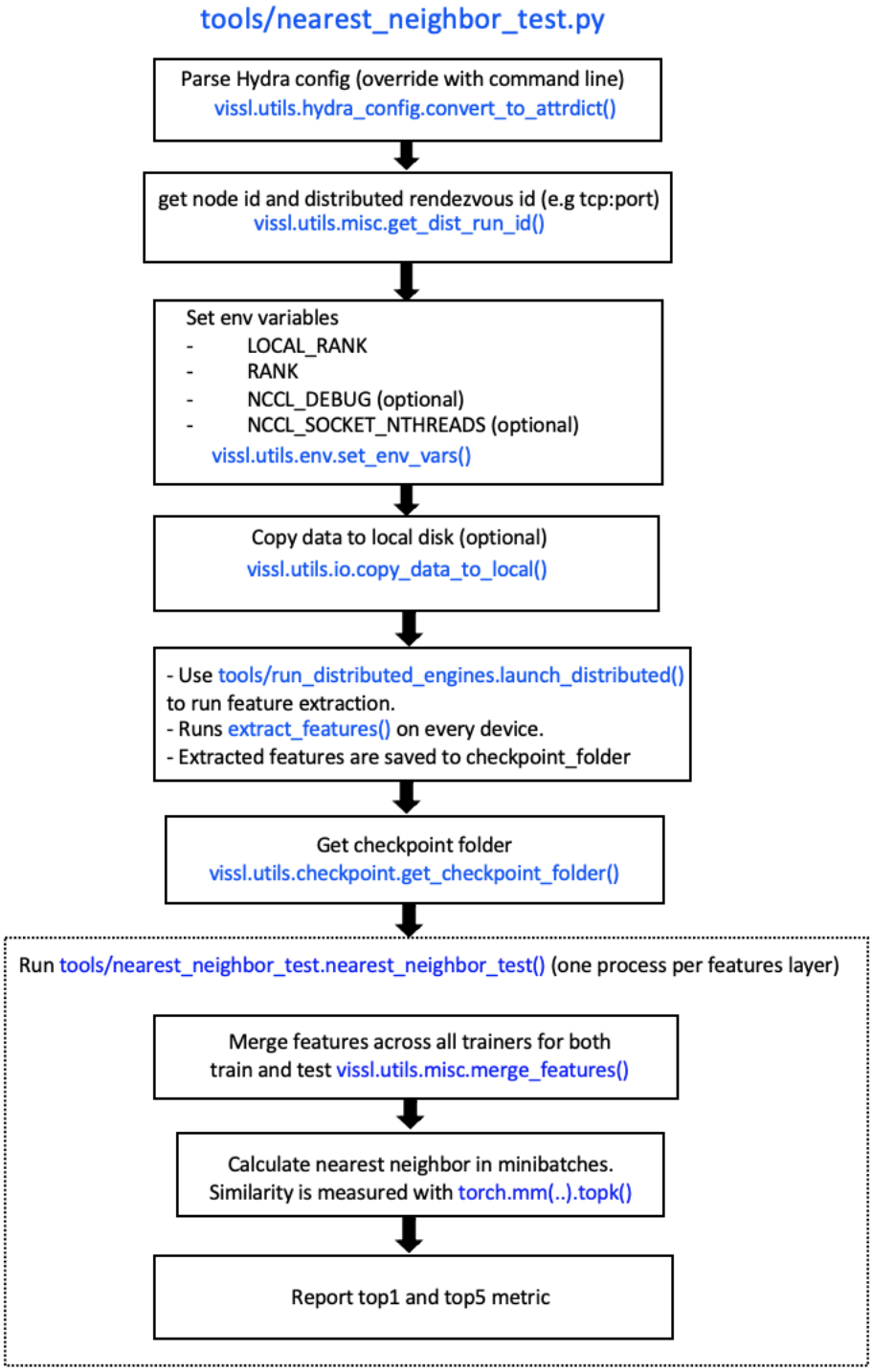

tools/train_svm.py
Parse Hydra config (override with command line)
vissl.utils.hydra_config.convert_to_attrdict()
get node id and distributed rendezvous id (e.g tcp:port)
vissl.utils.misc.get_dist_run_id()
Set env variables
- LOCAL_RANK
- RANK
- NCCL_DEBUG (optional)
- NCCL_SOCKET_NTHREADS (optional)
vissl.utils.env.set_env_vars()
Copy data to local disk (optional)
vissl.utils.io.copy_data_to_local()
- Use tools/run_distributed_engines.launch_distributed()
to run feature extraction.
- Runs extract_features() on every device.
- Extracted features are saved to checkpoint_folder
Get checkpoint folder
vissl.utils.checkpoint.get_checkpoint_folder()
Multi-process execution of train_svm() (one process per features layer)
Merge features across all trainers
vissl.utils.misc.merge_features()
Create SVMTrainer object
vissl.utils.svm_utils.svm_trainer.SVMTrainer()
- Train SVM for all classes and
- Multi-threaded execution for a given class to train on all costs
- save cross-validated SVM models
vissl.utils.svm_utils.svm_trainer.SVMTrainer().train()
Test SVM and save AP file
vissl.utils.svm_utils.svm_trainer.SVMTrainer().test()

CHAPTER

TWELVE

NEAREST NEIGHBOR BENCHMARK: STEP-BY-STEP EXECUTION

We demonstrate step-by-step execution of Nearest Neighbor benchmark in VISSL in the following flowchart.


tools/nearest_neighbor_test.py
Parse Hydra config (override with command line)
vissl.utils.hydra_config.convert_to_attrdict()
get node id and distributed rendezvous id (e.g tcp:port)
vissl.utils.misc.get_dist_run_id()
Set env variables
- LOCAL_RANK
- RANK
- NCCL_DEBUG (optional)
- NCCL_SOCKET_NTHREADS (optional)
vissl.utils.env.set_env_vars()
Copy data to local disk (optional)
vissl.utils.io.copy_data_to_local()
- Use tools/run_distributed_engines.launch_distributed()
to run feature extraction.
- Runs extract_features() on every device.
- Extracted features are saved to checkpoint_folder
Get checkpoint folder
vissl.utils.checkpoint.get_checkpoint_folder()
Run tools/nearest_neighbor_test.nearest_neighbor_test() (one process per features layer)
Merge features across all trainers for both
train and test vissl.utils.misc.merge_features()
Calculate nearest neighbor in minibatches.
Similarity is measured with torch.mm(..).topk()
Report top1 and top5 metric

CHAPTER

THIRTEEN

TRAIN MODELS ON CPU

VISSL supports training any model on CPUs. Typically, this involves correctly setting the `MACHINE.DEVICE=cpu` and adjusting the distributed settings accordingly. For example, the config settings will look like:

```
MACHINE:
  DEVICE: cpu
DISTRIBUTED:
  BACKEND: gloo            # set to "gloo" for cpu only trianing
  NUM_NODES: 1             # no change needed
  NUM_PROC_PER_NODE: 2     # user sets this to number of gpus to use
  INIT_METHOD: tcp         # set to "file" if desired
  RUN_ID: auto             # Set to file_path if using file method. No change needed␣
↪for tcp and a free port on machine is automatically detected.
```

CHAPTER

FOURTEEN

TRAIN ANYTHING ON 1-GPU

If you have a configuration file (any vissl compatible file) for any training, that you want to run on 1-gpu only (for example: train SimCLR on 1 gpu, etc), you don't need to modify the config file. VISSL provides a helper script that takes care of all the adjustments. This can facilitate debugging by allowing users to insert `pdb` in their code. VISSL also takes care of auto-scaling the Learning rate for various schedules (cosine, multistep, step etc.) if you have enabled the auto_scaling (see `config.OPTIMIZER.param_schedulers.lr.auto_lr_scaling`). You can simply achieve this by using the `low_resource_1gpu_train_wrapper.sh` script. An example usage:

```
cd $HOME/vissl
./dev/low_resource_1gpu_train_wrapper.sh config=test/integration_test/quick_swav
```

CHAPTER

FIFTEEN

TRAIN ON SLURM CLUSTER

VISSL supports SLURM by default for training models. VISSL code automatically detects if the training environment is SLURM based on SLURM environment variables like `SLURM_NODEID`, `SLURMD_NODENAME`, `SLURM_STEP_NODELIST`.

VISSL also provides a helper bash script dev/launch_slurm.sh that allows launching a given training on SLURM. Users can modify this script to meet their needs.

The bash script takes the following inputs:

```
# number of machines to distribute training on
NODES=${NODES-1}
# number of gpus per machine to use for training
NUM_GPU=${NUM_GPU-8}
# gpus type: P100 | V100 | V100_32G etc. User should set this based on their machine
GPU_TYPE=${GPU_TYPE-V100}
# name of the training. for example: simclr_2node_resnet50_in1k. This is helpful to␣
↪clearly recognize the training
EXPT_NAME=${EXPT_NAME}
# how much CPU memory to use
MEM=${MEM-250g}
# number of CPUs used for each trainer (i.e. each gpu)
CPU=${CPU-8}
# directory where all the training artifacts like checkpoints etc will be written
OUTPUT_DIR=${OUTPUT_DIR}
# partition of the cluster on which training should run. User should determine this␣
↪parameter for their cluster
PARTITION=${PARTITION-learnfair}
# any helpful comment that slurm dashboard can display
COMMENT=${COMMENT-vissl_training}
GITHUB_REPO=${GITHUB_REPO-vissl}
# what branch of VISSL should be used. specify your custom branch
BRANCH=${BRANCH-master}
# automatically determined and used for distributed training.
# each training run must have a unique id and vissl defaults to date
RUN_ID=$(date +'%Y%m%d')
# number of dataloader workers to use per gpu
NUM_DATA_WORKERS=${NUM_DATA_WORKERS-8}
# multi-processing method to use in PyTorch. Options: forkserver | fork | spawn
MULTI_PROCESSING_METHOD=${MULTI_PROCESSING_METHOD-forkserver}
# specify the training configuration to run. For example: to train swav for 100epochs
# config=pretrain/swav/swav_8node_resnet config.OPTIMIZER.num_epochs=100
CFG=( "$@" )
```

To run the script for training SwAV on 8 machines where each machine has 8-gpus and for 100epochs, the script can be run as:

```
cd $HOME/vissl && NODES=8 \
  NUM_GPU=8 \
  GPU_TYPE=V100 \
  MEM=200g \
  CPU=8 \
  EXPT_NAME=swav_100ep_rn50_in1k \
  OUTPUT_DIR=/tmp/swav/ \
  PARTITION=learnfair \
  BRANCH=master \
  NUM_DATA_WORKERS=4 \
  MULTI_PROCESSING_METHOD=forkserver \
  ./dev/launch_slurm.sh \
  config=pretrain/swav/swav_8node_resnet config.OPTIMIZER.num_epochs=100
```

CHAPTER

SIXTEEN

TRAIN ROTNET MODEL

VISSL reproduces the self-supervised approach **Unsupervised Representation Learning by Predicting Image Rotations** proposed by **Spyros Gidaris, Praveer Singh, Nikos Komodakis** in https://arxiv.org/abs/1803.07728.

16.1 How to train RotNet model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset using 4 rotation angles:

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet
```

16.1.1 Training different model architecture

VISSL supports many backbone architectures including AlexNet, ResNe(X)ts. Some examples below:

- **Train AlexNet model**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet \
    config.MODEL.TRUNK.NAME=alexnet_rotnet
```

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

16.1.2 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the RotNet model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

16.2 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using RotNet approach and the benchmarks.

16.3 Citation

```
@misc{gidaris2018unsupervised,
    title={Unsupervised Representation Learning by Predicting Image Rotations},
    author={Spyros Gidaris and Praveer Singh and Nikos Komodakis},
    year={2018},
    eprint={1803.07728},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

SEVENTEEN

TRAIN JIGSAW MODEL

VISSL reproduces the self-supervised approach **Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles** proposed by **Mehdi Noroozi and Paolo Favaro** in https://arxiv.org/abs/1603.09246.

17.1 How to train Jigsaw model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset using 2000 permutations.

```
python tools/run_distributed_engines.py config=pretrain/jigsaw/jigsaw_8gpu_resnet
```

17.1.1 Training with different permutations

In order to adjust the permutations and retrain, you can do so from command line. For example: to train for 10K permutations instead, VISSL provides the configuration files with necessary changes related to 10K permutation. Run:

```
python tools/run_distributed_engines.py config=pretrain/jigsaw/jigsaw_8gpu_resnet \
    +config/pretrain/jigsaw/permutations=perm10K
```

Similarly, you can train for 100 permutations and create new config files for a different permutations settings following the above configs as examples.

17.1.2 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the Jigsaw model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/jigsaw/jigsaw_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/jigsaw/jigsaw_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

17.2 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using Jigsaw approach and the benchmarks.

17.3 Permutations

Following Goyal et al we use the exact permutation files for Jigsaw training available here and refer users to directly use the files from the above source.

17.4 Citation

```
@misc{noroozi2017unsupervised,
    title={Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles},
    author={Mehdi Noroozi and Paolo Favaro},
    year={2017},
    eprint={1603.09246},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

EIGHTEEN

TRAIN NPID (AND NPID++) MODEL

VISSL reproduces the self-supervised approach **Unsupervised Feature Learning via Non-Parametric Instance Discrimination** proposed by **Zhirong Wu, Yuanjun Xiong, Stella Yu, Dahua Lin** in this paper. The NPID baselines were improved further by **Misra et. al** in **Self-Supervised Learning of Pretext-Invariant Representations** proposed in this paper.

18.1 How to train NPID model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset with NPID approach using 4,096 negatives selected randomly and feature projection dimension 128:

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet
```

18.2 How to Train NPID++ model

To train the NPID++ baselines with a ResNet-50 on ImageNet with 32000 negatives, 800 epochs and 4 machines (32-gpus) as in the PIRL paper:

```
python tools/run_distributed_engines.py config=pretrain/npid/npid++_4nodes_resnet
```

18.2.1 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different momentum value (say 0.99) for memory and different temperature 0.05 for logits, using 16000 negatives, the NPID training command would look like:

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet \
    config.LOSS.nce_loss_with_memory.temperature=0.05 \
    config.LOSS.nce_loss_with_memory.memory_params.momentum=0.99 \
    config.LOSS.nce_loss_with_memory.negative_sampling_params.num_negatives=16000
```

The full set of loss params that VISSL allows modifying:

```
nce_loss_with_memory:
  # setting below to "cross_entropy" yields the InfoNCE loss
  loss_type: "nce"
  norm_embedding: True
  temperature: 0.07
  # if the NCE loss is computed between multiple pairs, we can set a loss weight per␣
↪term
  # can be used to weight different pair contributions differently.
  loss_weights: [1.0]
  norm_constant: -1
  update_mem_with_emb_index: -100
  negative_sampling_params:
    num_negatives: 16000
    type: "random"
  memory_params:
    memory_size: -1
    embedding_dim: 128
    momentum: 0.5
    norm_init: True
    update_mem_on_forward: True
  # following parameters are auto-filled before the loss is created.
  num_train_samples: -1    # @auto-filled
```

18.2.2 Training different model architecture

VISSL supports many backbone architectures including AlexNet, ResNe(X)ts. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

18.2.3 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the NPID model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

18.3 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using NPID and NPID++ approach and the benchmarks.

18.4 Citations

- **NPID**

```
@misc{wu2018unsupervised,
    title={Unsupervised Feature Learning via Non-Parametric Instance-level␣
↪Discrimination},
    author={Zhirong Wu and Yuanjun Xiong and Stella Yu and Dahua Lin},
    year={2018},
    eprint={1805.01978},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

- **NPID++**

```
@misc{misra2019selfsupervised,
    title={Self-Supervised Learning of Pretext-Invariant Representations},
    author={Ishan Misra and Laurens van der Maaten},
    year={2019},
    eprint={1912.01991},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

NINETEEN

TRAIN CLUSTERFIT MODEL

VISSL reproduces the self-supervised approach **ClusterFit: Improving Generalization of Visual Representations** proposed by **Xueting Yan, Ishan Misra, Abhinav Gupta, Deepti Ghadiyaram, Dhruv Mahajan** in this paper.

19.1 How to train ClusterFit model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including data augmentations, collators etc required for this approach.

ClusterFit approach involves 2 steps:

- **Step1**: Using a pre-trained model (could be trained any way), the features are extracted on the training dataset (like ImageNet). The extracted features are clustered via k-means into N clusters (for example: 16000 clusters). For faster clustering, libraries like FAISS can be used (supported in VISSL). The cluster centroids are treated as the labels for the images and used for training in the next step.
- **Step2**: The model is trained (scratch initialization) but using the labels generated in Step 1.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset and using RotNet model to extract features:

```
# Step1: Extract features
python tools/run_distributed_engines.py config=pretrain/clusterfit/cluster_features_
↪resnet_8gpu_rotation_in1k \
    config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<vissl_compatible_weights.torch>

# Step2: Train clusterFit model
python tools/run_distributed_engines.py config=pretrain/clusterfit/clusterfit_resnet_
↪8gpu_imagenet \
    config.DATA.TRAIN.LABEL_PATHS=[<labels_file_from_step1.npy>]
```

The full set of hyperparams supported by VISSL for ClusterFit Step-1 include:

```
CLUSTERFIT:
  NUM_CLUSTERS: 16000
  # currently we only support faiss backend for clustering.
  CLUSTER_BACKEND: faiss
  # how many iterations to use for faiss
  N_ITER: 50
  FEATURES:
    DATA_PARTITION: TRAIN
    DATASET_NAME: imagenet1k
    LAYER_NAME: res5
```

19.1.1 How to use other pre-trained models in VISSL

VISSL supports Torchvision models out of the box. Generally, for loading any non-VISSL model, one needs to correctly set the following configuration options:

```
WEIGHTS_INIT:
  # path to the .torch weights files
  PARAMS_FILE: ""
  # name of the state dict. checkpoint = {"classy_state_dict": {layername:value}}.␣
↪Options:
  #   1. classy_state_dict - if model is trained and checkpointed with VISSL.
  #      checkpoint = {"classy_state_dict": {layername:value}}
  #   2. "" - if the model_file is not a nested dictionary for model weights i.e.
  #      checkpoint = {layername:value}
  #   3. key name that your model checkpoint uses for state_dict key name.
  #      checkpoint = {"your_key_name": {layername:value}}
  STATE_DICT_KEY_NAME: "classy_state_dict"
  # specify what layer should not be loaded. Layer names with this key are not copied
  # By default, set to BatchNorm stats "num_batches_tracked" to be skipped.
  SKIP_LAYERS: ["num_batches_tracked"]
  ####### If loading a non-VISSL trained model, set the following two args carefully #
↪########
  # to make the checkpoint compatible with VISSL, if you need to remove some names
  # from the checkpoint keys, specify the name
  REMOVE_PREFIX: ""
  # In order to load the model (if not trained with VISSL) with VISSL, there are 2␣
↪scenarios:
  #    1. If you are interested in evaluating the model features and freeze the trunk.
  #       Set APPEND_PREFIX="trunk.base_model." This assumes that your model is␣
↪compatible
  #       with the VISSL trunks. The VISSL trunks start with "_feature_blocks."␣
↪prefix. If
  #       your model doesn't have these prefix you can append them. For example:
  #       For TorchVision ResNet trunk, set APPEND_PREFIX="trunk.base_model._feature_
↪blocks."
  #    2. where you want to load the model simply and finetune the full model.
  #       Set APPEND_PREFIX="trunk."
  #       This assumes that your model is compatible with the VISSL trunks. The VISSL
  #       trunks start with "_feature_blocks." prefix. If your model doesn't have␣
↪these
  #       prefix you can append them.
  #       For TorchVision ResNet trunk, set APPEND_PREFIX="trunk._feature_blocks."
  # NOTE: the prefix is appended to all the layers in the model
  APPEND_PREFIX: ""
```

19.1.2 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the RotNet model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet␣
↪config.DISTRIBUTED.NUM_PROC_PER_NODE=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/rotnet/rotnet_8gpu_resnet␣
↪config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

19.2 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using RotNet approach and the benchmarks.

19.3 Citation

```
@misc{yan2019clusterfit,
  title={ClusterFit: Improving Generalization of Visual Representations},
  author={Xueting Yan and Ishan Misra and Abhinav Gupta and Deepti Ghadiyaram and␣
↪Dhruv Mahajan},
  year={2019},
  eprint={1912.03330},
  archivePrefix={arXiv},
  primaryClass={cs.CV}
}
```

CHAPTER

TWENTY

TRAIN PIRL MODEL

Author: imisra@fb.com

VISSL reproduces the self-supervised approach **Self-Supervised Learning of Pretext-Invariant Representations** proposed by **Ishan Misra and Laurens van der Maaten** in this paper.

20.1 How to train PIRL model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 4-machines (8-nodes) on ImageNet-1K dataset with PIRL approach using 32,000 negatives selected randomly and feature projection dimension 128:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50
```

20.2 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different momentum value (say 0.99) for memory and different temperature 0.05 for logits, using 16000 negatives, the NPID training command would look like:

```
python tools/run_distributed_engines.py config=pretrain/npid/npid_8gpu_resnet \
    config.LOSS.nce_loss_with_memory.temperature=0.05 \
    config.LOSS.nce_loss_with_memory.memory_params.momentum=0.99 \
    config.LOSS.nce_loss_with_memory.negative_sampling_params.num_negatives=16000
```

The full set of loss params that VISSL allows modifying:

```
nce_loss_with_memory:
  # setting below to "cross_entropy" yields the InfoNCE loss
  loss_type: "nce"
  norm_embedding: True
  temperature: 0.07
  # if the NCE loss is computed between multiple pairs, we can set a loss weight per␣
↪term
  # can be used to weight different pair contributions differently.
  loss_weights: [1.0]
  norm_constant: -1
```

(continues on next page)

(continued from previous page)

```
  update_mem_with_emb_index: -100
  negative_sampling_params:
    num_negatives: 16000
    type: "random"
  memory_params:
    memory_size: -1
    embedding_dim: 128
    momentum: 0.5
    norm_init: True
    update_mem_on_forward: True
  # following parameters are auto-filled before the loss is created.
  num_train_samples: -1    # @auto-filled
```

20.3 Training different model architecture

VISSL supports many backbone architectures including ResNe(X)ts, wider ResNets. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

- **Train ResNet-50-w2 (2x wider):**

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101 \
    config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.WIDTH_MULTIPLIER=2
```

20.4 Training with Gaussian Blur augmentation

Gaussian Blur augmentation has being a crucial transformation for better performance in approaches like SimCLR, SwAV, etc. The original PIRL method didn't use Gaussian Blur augmentation however PIRL author (imisra@fb.com) provide configuration for how to use the Gaussian Blur for training PIRL models. The command to run:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    +config/pretrain/pirl/transforms=photo_gblur
```

Please consult the *photo_gblur.yaml* config for the transformation composition.

20.5 Training with MLP head

Recent self-supervised approaches like SimCLR, MoCo, SwAV have benefitted significantly from using an MLP head. Original PIRL work didn't use MLP head but PIRL author (imisra@fb.com) provide configuration for using MLP head in PIRL and also open source the models. The command to run:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    +config/pretrain/pirl/models=resnet50_mlphead
```

Similarly, to train a ResNet-50-w2 (ie. 2x wider ResNet-50) with PIRL using MLP head:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    +config/pretrain/pirl/models=resnet50_w2_mlphead
```

Similarly, to train a ResNet-50-w4 (ie. 4x wider ResNet-50) with PIRL using MLP head:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    +config/pretrain/pirl/models=resnet50_w4_mlphead
```

20.6 Vary the number of epochs

In order to vary the number of epochs to use for training PIRL models, one can achieve this simply from command line. For example, to train the PIRL model for 100 epochs instead, pass the *num_epochs* parameter from command line:

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    config.OPTIMIZER.num_epochs=100
```

20.7 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the PIRL model on 8-gpus or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1 config.DISTRIBUTED.NUM_NODES=1
```

- **Training on 8-gpus:**

```
python tools/run_distributed_engines.py config=pretrain/pirl/pirl_jigsaw_4node_
↪resnet50 \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=1
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

20.8 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL for PIRL and the benchmarks.

20.9 Citations

```
@misc{misra2019selfsupervised,
    title={Self-Supervised Learning of Pretext-Invariant Representations},
    author={Ishan Misra and Laurens van der Maaten},
    year={2019},
    eprint={1912.01991},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

TWENTYONE

TRAIN SIMCLR MODEL

VISSL reproduces the self-supervised approach **A Simple Framework for Contrastive Learning of Visual Representations** proposed by **Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton** in this paper.

21.1 How to train SimCLR model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 4-machines (8-nodes) on ImageNet-1K dataset with SimCLR approach using MLP-head, loss temperature of 0.1 and feature projection dimension 128:

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet
```

21.2 Using Synchronized BatchNorm for training

For training SimCLR models, we convert all the BatchNorm layers to Global BatchNorm. For this, VISSL supports PyTorch SyncBatchNorm module and NVIDIA's Apex SyncBatchNorm layers. Set the config params `MODEL.SYNC_BN_CONFIG.SYNC_BN_TYPE` to `apex` or `pytorch`.

If you want to use Apex, VISSL provides `anaconda` and `pip` packages of Apex (compiled with Optimzed C++ extensions/CUDA kernels). The Apex packages are provided for all versions of `CUDA (9.2, 10.0, 10.1, 10.2, 11.0), PyTorch >= 1.4 and Python >=3.6 and <=3.9`.

To use SyncBN during training, one needs to set the following parameters in configuration file:

```
MODEL:
  SYNC_BN_CONFIG:
    CONVERT_BN_TO_SYNC_BN: True
    SYNC_BN_TYPE: apex
    # 1) if group_size=-1 -> use the VISSL default setting. We synchronize within a
    #     machine and hence will set group_size=num_gpus per node. This gives the best
    #     speedup.
    # 2) if group_size>0 -> will set group_size=value set by user.
    # 3) if group_size=0 -> no groups are created and process_group=None. This means
    #     global sync is done.
    GROUP_SIZE: 8
```

21.3 Using LARC for training

SimCLR training uses LARC from NVIDIA's Apex LARC. To use LARC, users need to set config option `OPTIMIZER.use_larc=True`. VISSL exposed LARC parameters that users can tune. Full list of LARC parameters exposed by VISSL:

```
OPTIMIZER:
  name: "sgd"
  use_larc: False  # supported for SGD only for now
  larc_config:
    clip: False
    eps: 1e-08
    trust_coefficient: 0.001
```

Note: LARC is currently supported for SGD optimizer only.

21.4 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different temperature 0.2 for logits and different output projection dimension of 256:

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.LOSS.simclr_info_nce_loss.temperature=0.2 \
    config.LOSS.simclr_info_nce_loss.buffer_params.embedding_dim=256
```

The full set of loss params that VISSL allows modifying:

```
simclr_info_nce_loss:
  temperature: 0.1
  buffer_params:
    embedding_dim: 128
    world_size: 64                # automatically inferred
    effective_batch_size: 4096    # automatically inferred
```

21.5 Training different model architecture

VISSL supports many backbone architectures including ResNe(X)ts, wider ResNets. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

- **Train ResNet-50-w2 (2x wider):**

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101 \
    config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.WIDTH_MULTIPLIER=2
```

21.6 Training with Multi-Crop data augmentation

The original SimCLR approach is proposed for 2 positives per image. We expand the SimCLR approach to work for more positives following the multi-crop augmentation proposed in SwAV paper. See SwAV paper https://arxiv.org/abs/2006.09882 for the multi-crop augmentation details.

Multi-crop augmentation can allow using more positives and also positives of different resolutions for SimCLR. VISSL provides a version of SimCLR loss for multi-crop training `multicrop_simclr_info_nce_loss`. In order to train SimCLR with multi-crop augmentation say crops `2x160 + 4x96` i.e. 2 crops of resolution 160 and 4 crops of resolution 96, the training command looks like:

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    +config/pretrain/simclr/transforms=multicrop_2x160_4x96
```

The `multicrop_2x160_4x96.yaml` configuration file changes 2 things:

- Transforms: Simply replace the `ImgReplicatePil` transform (which creates 2 copies of image) with `ImgPilToMultiCrop` which creates multi-crops of multiple resolutions.
- Loss: Use the loss `multicrop_simclr_info_nce_loss` instead which inherits from `simclr_info_nce_loss` and modifies the loss to work for multi-crop input.

21.6.1 Varying the multi-crop augmentation settings

VISSL allows modifying the crops to use. Full settings exposed:

```
TRANSFORMS:
  - name: ImgPilToMultiCrop
    total_num_crops: 6                      # Total number of crops to extract
    num_crops: [2, 4]                       # Specifies the number of type of crops.
    size_crops: [160, 96]                   # Specifies the height (height = width)␣
↪of each patch
    crop_scales: [[0.08, 1], [0.05, 0.14]]  # Scale of the crop
```

21.6.2 Varying the multi-crop loss settings

The full set of loss params that VISSL allows modifying:

```
multicrop_simclr_info_nce_loss:
  temperature: 0.1
  num_crops: 2                      # automatically inferred from data transforms
  buffer_params:
    world_size: 64                  # automatically inferred
    embedding_dim: 128
    effective_batch_size: 4096      # automatically inferred
```

21.7 Training with different MLP head

Original SimCLR approach used 2-layer MLP head. VISSL allows attaching any different desired head. In order to modify the MLP head (more layers, different dimensions etc), see the following examples:

- **3-layer MLP head:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
```

- **Use 2-layer MLP with hidden dimension 4096:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 4096], "use_relu": True}],
      ["mlp", {"dims": [4096, 128]}],
    ]
```

21.8 Vary the number of epochs

In order to vary the number of epochs to use for training SimCLR models, one can achieve this simply from command line. For example, to train the SimCLR model for 100 epochs instead, pass the `num_epochs` parameter from command line:

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.OPTIMIZER.num_epochs=100
```

21.9 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the SimCLR model on 8-gpus or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1 config.DISTRIBUTED.NUM_NODES=1
```

- **Training on 8-gpus:**

```
python tools/run_distributed_engines.py config=pretrain/simclr/simclr_8node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=1
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

21.10 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL for SimCLR and the benchmarks.

21.11 Citations

```
@misc{chen2020simple,
    title={A Simple Framework for Contrastive Learning of Visual Representations},
    author={Ting Chen and Simon Kornblith and Mohammad Norouzi and Geoffrey Hinton},
    year={2020},
    eprint={2002.05709},
    archivePrefix={arXiv},
    primaryClass={cs.LG}
}
```

CHAPTER

TWENTYTWO

TRAIN MOCO MODEL

Author: lefaudeux@fb.com

VISSL reproduces the self-supervised approach MoCo **Momentum Contrast for Unsupervised Visual Representation Learning** proposed by **Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, Ross Girshick** in this paper. The MoCo baselines were improved further by **Xinlei Chen, Haoqi Fan, Ross Girshick, Kaiming He** in "Improved Baselines with Momentum Contrastive Learning" proposed in this paper.

VISSL closely follows the implementation provided by MoCo authors themselves https://github.com/facebookresearch/moco.

22.1 How to train MoCo (and MoCo v2 model) model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset with MoCo-v2 approach using feature projection dimension 128:

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet
```

By default, VISSL provides configuration file for MoCo-v2 model as this has better baselines numbers. To train MoCo baseline instead, users should make 2 changes to the moco configuration file:

- change the `config.DATA.TRAIN.TRANSFORMS` by removing the `ImgPilGaussianBlur` transform.
- change the `config.MODEL.HEAD.PARAMS=[["mlp", {"dims": [2048, 128]}]]` i.e. replace the MLP-head with fc-head.

22.1.1 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different momentum value (say 0.99) for memory and different temperature 0.5 for logits, the MoCo training command would look like:

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet \
    config.LOSS.moco_loss.temperature=0.5 \
    config.LOSS.moco_loss.momentum=0.99
```

The full set of loss params that VISSL allows modifying:

```
moco_loss:
  embedding_dim: 128
  queue_size: 65536
  momentum: 0.999
  temperature: 0.2
```

22.1.2 Training different model architecture

VISSL supports many backbone architectures including ResNe(X)ts, wider ResNets. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

- **Train ResNet-50-w2 (2x wider ResNet-50):**

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=50 \
    config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.WIDTH_MULTIPLIER=2
```

22.1.3 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the MoCo model on 4 machines (32-gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/moco/moco_1node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus. However, MoCo doesn't work very well with this rule as per the authors in the paper.

Note: If you change the number of gpus for MoCo training, MoCo models require longer training in order to reproduce results. Hence, we recommend users to consult MoCo paper.

22.2 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using MoCo-v2 approach and the benchmarks.

22.3 Citations

- **MoCo**

```
@misc{he2020momentum,
    title={Momentum Contrast for Unsupervised Visual Representation Learning},
    author={Kaiming He and Haoqi Fan and Yuxin Wu and Saining Xie and Ross Girshick},
    year={2020},
    eprint={1911.05722},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

- **MoCo-v2**

```
@misc{chen2020improved,
    title={Improved Baselines with Momentum Contrastive Learning},
    author={Xinlei Chen and Haoqi Fan and Ross Girshick and Kaiming He},
    year={2020},
    eprint={2003.04297},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

TWENTYTHREE

TRAIN DEEPCLUSTER V2 MODEL

Author: mathilde@fb.com

VISSL reproduces the self-supervised approach called `DeepClusterV2` which is an improved version of original DeepCluster approach. The DeepClusterV2 approach was proposed in work **Unsupervised learning of visual features by contrasting cluster assignments** by **Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, Armand Joulin** in this paper. DeepClusterV2 combines the benefits of DeepCluster and NPID approaches.

23.1 How to train DeepClusterV2 model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset using feature projection dimension 128 for memory:

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet
```

23.2 Using Synchronized BatchNorm for training

For training DeepClusterV2 models, we convert all the BatchNorm layers to Global BatchNorm. For this, VISSL supports PyTorch SyncBatchNorm module and NVIDIA's Apex SyncBatchNorm layers. Set the config params `MODEL.SYNC_BN_CONFIG.SYNC_BN_TYPE` to `apex` or `pytorch`.

If you want to use Apex, VISSL provides `anaconda` and `pip` packages of Apex (compiled with Optimzed C++ extensions/CUDA kernels). The Apex packages are provided for all versions of `CUDA (9.2, 10.0, 10.1, 10.2, 11.0), PyTorch >= 1.4 and Python >=3.6 and <=3.9`.

To use SyncBN during training, one needs to set the following parameters in configuration file:

```
MODEL:
  SYNC_BN_CONFIG:
    CONVERT_BN_TO_SYNC_BN: True
    SYNC_BN_TYPE: apex
    # 1) if group_size=-1 -> use the VISSL default setting. We synchronize within a
    #     machine and hence will set group_size=num_gpus per node. This gives the best
    #     speedup.
    # 2) if group_size>0 -> will set group_size=value set by user.
    # 3) if group_size=0 -> no groups are created and process_group=None. This means
```

(continues on next page)

(continued from previous page)

```
    #     global sync is done.
    GROUP_SIZE: 8
```

23.3 Using Mixed Precision for training

DeepClusterV2 approach leverages mixed precision training by default for better training speed and reducing the model memory requirement. For this, we use NVIDIA Apex Library with Apex AMP level O1.

To use Mixed precision training, one needs to set the following parameters in configuration file:

```
MODEL:
  AMP_PARAMS:
    USE_AMP: True
    # Use O1 as it is robust and stable than O3. If you want to use O3, we recommend
    # the following setting:
    # {"opt_level": "O3", "keep_batchnorm_fp32": True, "master_weights": True, "loss_
↪scale": "dynamic"}
    AMP_ARGS: {"opt_level": "O1"}
```

23.4 Using LARC for training

DeepClusterV2 training uses LARC from NVIDIA's Apex LARC. To use LARC, users need to set config option `OPTIMIZER.use_larc=True`. VISSL exposed LARC parameters that users can tune. Full list of LARC parameters exposed by VISSL:

```
OPTIMIZER:
  name: "sgd"
  use_larc: False  # supported for SGD only for now
  larc_config:
    clip: False
    eps: 1e-08
    trust_coefficient: 0.001
```

Note: LARC is currently supported for SGD optimizer only.

23.5 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different temperature 0.2 for logits, projection dimension 256 for the embedding, the training command would look like:

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.LOSS.deepclusterv2_loss.temperature=0.2 \
    config.LOSS.deepclusterv2_loss.memory_params.embedding_dim=256
```

The full set of loss params that VISSL allows modifying:

```
deepclusterv2_loss:
  DROP_LAST: True               # automatically inferred from DATA.TRAIN.DROP_LAST
  BATCHSIZE_PER_REPLICA: 256    # automatically inferred from DATA.TRAIN.BATCHSIZE_PER_
↪REPLICA
  num_crops: 2                  # automatically inferred from DATA.TRAIN.TRANSFORMS
  temperature: 0.1
  num_clusters: [3000, 3000, 3000]
  kmeans_iters: 10
  memory_params:
    crops_for_mb: [0]
    embedding_dim: 128
  # following parameters are auto-filled before the loss is created.
  num_train_samples: -1         # automatically inferred
```

23.6 Training different model architecture

VISSL supports many backbone architectures including ResNe(X)ts, wider ResNets. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

- **Train ResNet-50-w2 (2x wider):**

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101 \
    config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.WIDTH_MULTIPLIER=2
```

23.7 Training with Multi-Crop data augmentation

DeepClusterV2 can be trained for for more positives following the multi-crop augmentation proposed in SwAV paper. See SwAV paper https://arxiv.org/abs/2006.09882 for the multi-crop augmentation details.

Multi-crop augmentation can allow using more positives and also positives of different resolutions. In order to train DeepClusterV2 with multi-crop augmentation say crops `2x160 + 4x96` i.e. 2 crops of resolution 160 and 4 crops of resolution 96, the training command looks like:

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    +config/pretrain/deepcluster_v2/transforms=multicrop_2x160_4x96
```

The `multicrop_2x160_4x96.yaml` configuration file changes the number of crop settings to 6 crops.

23.7.1 Varying the multi-crop augmentation settings

VISSL allows modifying the crops to use. Full settings exposed:

```
TRANSFORMS:
  - name: ImgPilToMultiCrop
    total_num_crops: 6                      # Total number of crops to extract
    num_crops: [2, 4]                       # Specifies the number of type of crops.
    size_crops: [160, 96]                   # Specifies the height (height = width)␣
↪of each patch
    crop_scales: [[0.08, 1], [0.05, 0.14]]  # Scale of the crop
```

23.8 Training with different MLP head

By default, the original DeepClusterV2 approach used the 2-layer MLP-head similar to SimCLR approach. VISSL allows attaching any different desired head. In order to modify the MLP head (more layers, different dimensions etc), see the following examples:

- **3-layer MLP head:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
```

- **Use 2-layer MLP with hidden dimension 4096:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 4096], "use_relu": True}],
      ["mlp", {"dims": [4096, 128]}],
    ]
```

23.9 Vary the number of epochs

In order to vary the number of epochs to use for training DeepClusterV2 models, one can achieve this simply from command line. For example, to train the DeepClusterV2 model for 100 epochs instead, pass the `num_epochs` parameter from command line:

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.OPTIMIZER.num_epochs=100
```

23.10 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the DeepClusterV2 model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1 config.DISTRIBUTED.NUM_NODES=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/deepcluster_v2/deepclusterv2_
↪2crops_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

23.11 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with VISSL using DeepClusterV2 approach and the benchmarks.

23.12 Citations

- **DeepClusterV2**

```
@misc{caron2020unsupervised,
    title={Unsupervised Learning of Visual Features by Contrasting Cluster␣
↪Assignments},
    author={Mathilde Caron and Ishan Misra and Julien Mairal and Priya Goyal and␣
↪Piotr Bojanowski and Armand Joulin},
    year={2020},
    eprint={2006.09882},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

TWENTYFOUR

TRAIN SWAV MODEL

Author: mathilde@fb.com

VISSL reproduces the self-supervised approach called `SwAV` **Unsupervised learning of visual features by contrasting cluster assignments** which was proposed by **Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, Armand Joulin** in this paper. SwAV clusters the features while enforcing consistency between cluster assignments produced for different augmentations (or "views") of the same image, instead of comparing features directly as in contrastive learning.

24.1 How to train SwAV model

VISSL provides yaml configuration file containing the exact hyperparam settings to reproduce the model. VISSL implements all the components including loss, data augmentations, collators etc required for this approach.

To train ResNet-50 model on 8-gpus on ImageNet-1K dataset using feature projection dimension 128 for memory:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet
```

24.2 Using Synchronized BatchNorm for training

For training SwAV models, we convert all the BatchNorm layers to Global BatchNorm. For this, VISSL supports PyTorch SyncBatchNorm module and NVIDIA's Apex SyncBatchNorm layers. Set the config params `MODEL.SYNC_BN_CONFIG.SYNC_BN_TYPE` to `apex` or `pytorch`.

If you want to use Apex, VISSL provides `anaconda` and `pip` packages of Apex (compiled with Optimzed C++ extensions/CUDA kernels). The Apex packages are provided for all versions of `CUDA (9.2, 10.0, 10.1, 10.2, 11.0), PyTorch >= 1.4 and Python >=3.6 and <=3.9`.

To use SyncBN during training, one needs to set the following parameters in configuration file:

```
MODEL:
  SYNC_BN_CONFIG:
    CONVERT_BN_TO_SYNC_BN: True
    SYNC_BN_TYPE: apex
    # 1) if group_size=-1 -> use the VISSL default setting. We synchronize within a
    #     machine and hence will set group_size=num_gpus per node. This gives the best
    #     speedup.
    # 2) if group_size>0 -> will set group_size=value set by user.
    # 3) if group_size=0 -> no groups are created and process_group=None. This means
    #     global sync is done.
    GROUP_SIZE: 8
```

24.3 Using Mixed Precision for training

SwAV approach leverages mixed precision training by default for better training speed and reducing the model memory requirement. For this, we use NVIDIA Apex Library with Apex AMP level O1.

To use Mixed precision training, one needs to set the following parameters in configuration file:

```
MODEL:
  AMP_PARAMS:
    USE_AMP: True
    # Use O1 as it is robust and stable than O3. If you want to use O3, we recommend
    # the following setting:
    # {"opt_level": "O3", "keep_batchnorm_fp32": True, "master_weights": True, "loss_
↪scale": "dynamic"}
    AMP_ARGS: {"opt_level": "O1"}
```

24.4 Using LARC for training

SwAV training uses LARC from NVIDIA's Apex LARC. To use LARC, users need to set config option `OPTIMIZER.use_larc=True`. VISSL exposed LARC parameters that users can tune. Full list of LARC parameters exposed by VISSL:

```
OPTIMIZER:
  name: "sgd"
  use_larc: False  # supported for SGD only for now
  larc_config:
    clip: False
    eps: 1e-08
    trust_coefficient: 0.001
```

Note: LARC is currently supported for SGD optimizer only.

24.5 Vary the training loss settings

Users can adjust several settings from command line to train the model with different hyperparams. For example: to use a different temperature 0.2 for logits, epsilon of 0.04, the training command would look like:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.LOSS.swav_loss.temperature=0.2 \
    config.LOSS.swav_loss.epsilon=0.04
```

The full set of loss params that VISSL allows modifying:

```
swav_loss:
  temperature: 0.1
  use_double_precision: False
  normalize_last_layer: True
  num_iters: 3
  epsilon: 0.05
  temp_hard_assignment_iters: 0
```

(continues on next page)

(continued from previous page)

```
  crops_for_assign: [0, 1]
  embedding_dim: 128              # automatically inferred from HEAD params
  num_crops: 2                    # automatically inferred from data transforms
  num_prototypes: [3000]          # automatically inferred from model HEAD settings
  # for dumping the debugging info in case loss becomes NaN
  output_dir: ""                  # automatically inferred and set to checkpoint dir
  queue:
    start_iter: 0
    queue_length: 0               # automatically adjusted to ensure queue_length %␣
↪global batch size = 0
    local_queue_length: 0         # automatically inferred to queue_length // world_size
```

24.6 Training different model architecture

VISSL supports many backbone architectures including ResNe(X)ts, wider ResNets. Some examples below:

- **Train ResNet-101:**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101
```

- **Train ResNet-50-w2 (2x wider):**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.MODEL.TRUNK.NAME=resnet config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.DEPTH=101 \
    config.MODEL.TRUNK.TRUNK_PARAMS.RESNETS.WIDTH_MULTIPLIER=2
```

- **Train RegNetY-400MF:**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.MODEL.TRUNK.NAME=regnet config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.name=regnet_
↪y_400mf
```

- **Train RegNetY-256GF:**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.MODEL.TRUNK.NAME=regnet \
    config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.depth=27 \
    config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.w_0=640 \
    config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.w_a=230.83 \
    config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.w_m=2.53 \
    config.MODEL.TRUNK.TRUNK_PARAMS.REGNET.group_width=373 \
    config.MODEL.HEAD.PARAMS=[["swav_head", {"dims": [10444, 10444, 128], "use_bn":␣
↪False, "num_clusters": [3000]}]]
```

24.7 Training with Multi-Crop data augmentation

SwAV is trained using the multi-crop augmentation proposed in SwAV paper.

Multi-crop augmentation can allow using more positives and also positives of different resolutions. In order to train SwAV with multi-crop augmentation say crops `2x224 + 4x96` i.e. 2 crops of resolution 224 and 4 crops of resolution 96, the training command looks like:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    +config/pretrain/swav/transforms=multicrop_2x224_4x96
```

The `multicrop_2x224_4x96.yaml` configuration file changes the number of crop settings to 6 crops and the right resolution.

24.7.1 Varying the multi-crop augmentation settings

VISSL allows modifying the crops to use. Full settings exposed:

```
TRANSFORMS:
  - name: ImgPilToMultiCrop
    total_num_crops: 6                      # Total number of crops to extract
    num_crops: [2, 4]                       # Specifies the number of type of crops.
    size_crops: [160, 96]                   # Specifies the height (height = width)␣
↪of each patch
    crop_scales: [[0.08, 1], [0.05, 0.14]]  # Scale of the crop
```

24.8 Training with different MLP head

By default, the original SwAV approach used the 2-layer MLP-head similar to SimCLR approach. VISSL allows attaching any different desired head. In order to modify the MLP head (more layers, different dimensions etc), see the following examples:

- **3-layer MLP head:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["swav_head", {"dims": [2048, 2048, 2048, 128], "use_bn": True, "num_clusters":␣
↪[3000]}],
    ]
```

- **Use 2-layer MLP with hidden dimension 4096:** Use the following head (example for ResNet model)

```
MODEL:
  HEAD:
    PARAMS: [
      ["swav_head", {"dims": [2048, 4096, 128], "use_bn": True, "num_clusters":␣
↪[3000]}],
    ]
```

24.9 Vary the number of epochs

In order to vary the number of epochs to use for training SwAV models, one can achieve this simply from command line. For example, to train the SwAV model for 100 epochs instead, pass the `num_epochs` parameter from command line:

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.OPTIMIZER.num_epochs=100
```

24.10 Vary the number of gpus

VISSL makes it extremely easy to vary the number of gpus to be used in training. For example: to train the SwAV model on 4 machines (32gpus) or 1gpu, the changes required are:

- **Training on 1-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=1 config.DISTRIBUTED.NUM_NODES=1
```

- **Training on 4 machines i.e. 32-gpu:**

```
python tools/run_distributed_engines.py config=pretrain/swav/swav_8node_resnet \
    config.DISTRIBUTED.NUM_PROC_PER_NODE=8 config.DISTRIBUTED.NUM_NODES=4
```

Note: Please adjust the learning rate following ImageNet in 1-Hour if you change the number of gpus.

24.11 Pre-trained models

See VISSL Model Zoo for the PyTorch pre-trained models with SwAV using DeepClusterV2 approach and the benchmarks.

24.12 Citations

- **DeepClusterV2**

```
@misc{caron2020unsupervised,
    title={Unsupervised Learning of Visual Features by Contrasting Cluster␣
↪Assignments},
    author={Mathilde Caron and Ishan Misra and Julien Mairal and Priya Goyal and␣
↪Piotr Bojanowski and Armand Joulin},
    year={2020},
    eprint={2006.09882},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

TWENTYFIVE

BENCHMARK: LINEAR IMAGE CLASSIFICATION

VISSL provides a standardized benchmark suite to evaluate the feature representation quality of self-supervised pre-trained models. A popular evaluation protocol is to freeze the model traink and train linear classifiers on several layers of the model on some target datasets (like ImageNet-1k, Places205, VOC07, iNaturalist2018). In VISSL, we support all the linear evals on all the datasets. We also provide standard set of hyperparams for various approaches in order to reproduce the model performance in SSL literature. For reproducibility, see VISSL Model Zoo.

Note: To run the benchmark, we recommend using the standard set of hyperparams provided by VISSL as these hyperparams reproduce results of large number of self-supervised approaches. Users are however free to modify the hyperparams to suit their evaluation criterion.

25.1 Eval Config settings using MLP head

Set the following in the config file to enable the feature evaluation

25.1.1 Attaching MLP head to many layers of trunk

- To attach linear classifier (`FC`) on the trunk output, example for a ResNet-50 model:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
        ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
    RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
        ["mlp", {"dims": [2048, 1000]}],
    ]
```

25.1.2 Attaching MLP head to trunk output

To attach a linear classifier at multiple layers of model following Zhang et. al style which has `BN -> FC` as the head, use `eval_mlp` head. For example, for a ResNet-50 model,

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["conv1", ["AvgPool2d", [[10, 10], 10, 4]]],
      ["res2", ["AvgPool2d", [[16, 16], 8, 0]]],
      ["res3", ["AvgPool2d", [[13, 13], 5, 0]]],
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
    RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
      ["eval_mlp", {"in_channels": 64, "dims": [9216, 1000]}],
      ["eval_mlp", {"in_channels": 256, "dims": [9216, 1000]}],
      ["eval_mlp", {"in_channels": 512, "dims": [8192, 1000]}],
      ["eval_mlp", {"in_channels": 1024, "dims": [9216, 1000]}],
      ["eval_mlp", {"in_channels": 2048, "dims": [8192, 1000]}],
      ["eval_mlp", {"in_channels": 2048, "dims": [2048, 1000]}],
    ]
```

25.2 Eval Config settings using SVM training

For SVM trainings, we only care about extracting the features from the model. We dump the features on disk and train SVMs. To extract the features:

25.2.1 Features from several layers of the trunk

For example, for a ResNet-50 model, to train features from many layers of the model, the example config:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True    # only extract the features and we will train␣
↪SVM on these
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
TRUNK:
```

(continues on next page)

(continued from previous page)

```
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

25.2.2 Features from the trunk output

For example, for a ResNet-50 model, to train features from model trunk output, the example config:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

Below, we provide instruction on how to run each benchmark.

25.3 Benchmark: ImageNet-1k

The configuration setting for this benchmark is provided here .

```
python tools/run_distributed_engines.py \
  config=benchmark/linear_image_classification/imagenet1k/eval_resnet_8gpu_transfer_
↪in1k_linear \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

25.4 Benchmark: Places205

The configuration setting for this benchmark is provided here .

```
python tools/run_distributed_engines.py \
  config=benchmark/linear_image_classification/places205/eval_resnet_8gpu_transfer_
↪places205_linear \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

25.5 Benchmark: iNaturalist2018

The configuration setting for this benchmark is provided here .

```
python tools/run_distributed_engines.py \
  config=benchmark/inear_image_classification/inaturalist18/eval_resnet_8gpu_transfer_
↪inaturalist18_linear \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

25.6 Benchmark: Linear SVM on VOC07

VISSL provides `train_svm.py` tool that will first extract features and then train/test SVMs on these features. The configuration setting for this benchmark is provided here .

```
python tools/train_svm.py \
  config=benchmark/inear_image_classification/voc07/eval_resnet_8gpu_transfer_voc07_
↪svm \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

Note: Please see VISSL documentation on how to run a given training on **1-gpu, multi-gpu or multi-machine**.

Note: Please see VISSL documentation on how to use the **builtin datasets**.

Note: Please see VISSL documentation on how to use YAML comfiguration system in VISSL to **override specific components like model** of a config file. For example, in the above file, user can replace ResNet-50 model with a different architecture like RegNetY-256 etc. easily.

CHAPTER

TWENTYSIX

BENCHMARK TASK: FULL-FINETUNING

Using a self-supervised model to initialize a network and further tune the weights on the target task is a very common evaluation protocol. This benchmark requires only initializing the model and no other settings in `MODEL.FEATURE_EVAL_SETTINGS` are needed unlike other benchmark tasks.

26.1 Benchmark: ImageNet-1k

VISSL provides the YAML configuration setting for this benchmark here which can be run as below.

```
python tools/run_distributed_engines.py \
  config=benchmark/imagenet1k_fulltune/eval_resnet_8gpu_transfer_in1k_fulltune \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

26.2 Benchmark: Places205

VISSL provides the YAML configuration setting for this benchmark here which can be run as below.

```
python tools/run_distributed_engines.py \
  config=benchmark/places205_fulltune/eval_resnet_8gpu_transfer_places205_fulltune \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

Note: Please see VISSL documentation on how to run a given training on **1-gpu, multi-gpu or multi-machine**.

Note: Please see VISSL documentation on how to use the **builtin datasets** if you want to run this benchmark on a different target task..

Note: Please see VISSL documentation on how to use YAML comfiguration system in VISSL to **override specific components like model** of a config file. For example, in the above file, user can replace ResNet-50 model with a different architecture like RegNetY-256 etc. easily.

CHAPTER

TWENTYSEVEN

BENCHMARK: NEAREST NEIGHBOR K-MEANS

VISSL supports Nearest Neighbor evaluation task using k-means. We closely follow the benchmark setup from Zhirong Wu et al. https://github.com/zhirongw/lemniscate.pytorch#nearest-neighbor. For the Nearest neighbor evaluation, the process involves 2 steps:

- **Step1**: Extract the relevant features from the model for both training and validation set.
- **Step2**: Perform k-means clustering and evaluation on these features

VISSL provides a dedicated tool `tools/nearest_neighbor_test.py` that performs both Step-1 and Step-2 above.

Note: To run the benchmark, we recommend using the standard set of hyperparams provided by VISSL as these hyperparams reproduce results of large number of self-supervised approaches. Users are however free to modify the hyperparams to suit their evaluation criterion.

27.1 Eval Config settings using MLP head

Set the following in the config file to enable the feature evaluation properly.

27.1.1 kNN on many layers of the trunk

For the Step1, if we want to extract features from many layers of the trunk, the config setting should look like below. For example for a ResNet-50 model:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True  # only freeze the trunk
    EXTRACT_TRUNK_FEATURES_ONLY: True   # we extract features from the trunk only
    SHOULD_FLATTEN_FEATS: False   # don't flatten the features and return as is
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

27.1.2 kNN on the trunk output

If we want to perform kNN only on the trunk output, the configuration setting should look like below. For example, for a ResNet-50 model:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True  # only freeze the trunk
    EXTRACT_TRUNK_FEATURES_ONLY: True   # we extract features from the trunk only
    SHOULD_FLATTEN_FEATS: False   # don't flatten the features and return as is
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

27.1.3 kNN on the model head output (self-supervised head)

For a given self-supervised approach, we want to perform kNN on the output of the model head. This is very common where the model head is a projection head and projects the trunk features into a low-dimensional space. The config settings should look like below. The example below is for SimCLR head + ResNet-50. Users can replace the `MODEL.HEAD.PARAMS` with the head settings used in the respective self-supervised model training.

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_AND_HEAD: True   # both head and trunk will be frozen (including BN␣
↪in eval mode)
    EVAL_TRUNK_AND_HEAD: True     # initialized the model head as well from weights
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    # SimCLR 2-layer model head structure
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
```

27.2 Benchmark: ImageNet-1k

VISSL provides configuration settings for the benchmark here.

To run the benchmark:

```
python tools/nearest_neighbor_test.py config=benchmark/nearest_neighbor/eval_resnet_
↪8gpu_in1k_kNN \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

27.3 Benchmark: Places205

VISSL provides configuration settings for the benchmark here.

To run the benchmark:

```
python tools/nearest_neighbor_test.py config=benchmark/nearest_neighbor/eval_resnet_
↪8gpu_places205_kNN \
  config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

Note: Please see VISSL documentation on how to run a given training on **1-gpu, multi-gpu or multi-machine**.

Note: Please see VISSL documentation on how to use the **builtin datasets**.

Note: Please see VISSL documentation on how to use YAML comfiguration system in VISSL to **override specific components like model** of a config file. For example, in the above file, user can replace ResNet-50 model with a different architecture like RegNetY-256 etc. easily.

CHAPTER

TWENTYEIGHT

BENCHMARK TASK: FULL FINETUNING ON IMAGENET 1% , 10% SUBSETS

Evaluating a self-supervised pre-trained model on the target dataset which represents 1% or 10% of Imagenet dataset has become a very common evaluation criterion. VISSL provides the benchmark settings for this benchmark here.

28.1 1% and 10% Data subsets

VISSL uses the 1% and 10% datasets from the SimCLR work. Users can download the datasets from here. Users can use the `DATA.TRAIN.DATA_SOURCES=[disk_filelist]` to load the images in these files. Users should replace each line with the valid full image path for that image. Users should also extract the labels out from these datasets using the `image_id` of each image.

Once the user has the valid image and labels files (.npy), users should set the dataset paths in VISSL dataset_catalog.json for the datasets `google-imagenet1k-per01` and `google-imagenet1k-per10`

28.2 Benchmark: 1% ImageNet

Users can run the benchmark on 1% ImageNet subsets from SimCLR with the following command:

```
python tools/run_distributed_engines.py \
  config=benchmark/semi_supervised/imagenet1k/eval_resnet_8gpu_transfer_in1k_semi_sup_
↪fulltune \
  +config/benchmark/semi_supervised/imagenet1k/dataset=simclr_in1k_per01
```

28.3 Benchmark: 10% ImageNet

Users can run the benchmark on 10% ImageNet subsets from SimCLR with the following command:

```
python tools/run_distributed_engines.py \
  config=benchmark/semi_supervised/imagenet1k/eval_resnet_8gpu_transfer_in1k_semi_sup_
↪fulltune \
  +config/benchmark/semi_supervised/imagenet1k/dataset=simclr_in1k_per10
```

Note: Please see VISSL documentation on how to run a given training on **1-gpu, multi-gpu or multi-machine**.

Note: Please see VISSL documentation on how to use the **builtin datasets** if you want to run this benchmark on a different target task..

Note: Please see VISSL documentation on how to use YAML comfiguration system in VISSL to **override specific components like model** of a config file. For example, in the above file, user can replace ResNet-50 model with a different architecture like RegNetY-256 etc. easily.

CHAPTER

TWENTYNINE

BENCHMARK TASK: OBJECT DETECTION

Object Detection is a very common benchmark for evaluating feature representation quality. In VISSL, we use Detectron2 for the object detection benchmark.

This benchmark involves 2 steps:

- **Step1**: Converting the self-supervised model weights so they are compatible with `Detectron2`.
- **Step2**: Using the converted weights in Step1, run the benchmark.

29.1 Converting weights to Detectron2

VISSL provides a script to convert the weight of VISSL compatible models to Detectron2. We recommend users to adapt this script to suit their needs (different model architecture etc).

To run the script, follow the command:

```
python extra_scripts/convert_vissl_to_detectron2.py \
  --input_model_file <input_model_path>.torch  \
  --output_model <converted_d2_model_path>.torch \
  --weights_type torch \
  --state_dict_key_name classy_state_dict
```

The script above converts ResNe(X)ts models in VISSL to the models compatible with ResNe(X)ts in Detectron2.

29.2 Benchmark: Faster R-CNN on VOC07

VISSL provides the YAML configuration files for `Detectron2` for the benchmark task of Object detection using `Faster R-CNN` on VOC07. The configuration files are available here. To run the benchmark, VISSL provides a python script that closely follows MoCo object detection.

Please make sure to install Detectron2 following the Detectron2 Installation instructions.

To run the benchmark:

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc07/rn50_transfer_
↪voc07_detectron2_e2e.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

Note: We recommend users to consult Detectron2 documentation for how to use the configuration files and how to run the trainings.

29.2.1 PIRL object detection

To reproduce the object detection benchmark, the LR and warmup iterations are different. Use the following command:

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc07/pirl_npid/rn50_
↪transfer_voc07_pirl_npid.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

29.3 Benchmark: Faster R-CNN on VOC07+12

VISSL provides the YAML configuration files for `Detectron2` for the benchmark task of Object detection using `Faster R-CNN` on VOC07+12. The configuration files are available here. To run the benchmark, VISSL provides a python script that closely follows MoCo object detection.

Please make sure to install Detectron2 following the Detectron2 Installation instructions.

For the VOC07+12 benchmark, most self-supervised approaches use their set of hyperparams. VISSL provides the settings used in

29.3.1 Scaling and Benchmarking Self-Supervised Visual Representation Learning

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc0712/iccv19/rn50_
↪transfer_voc0712_detectron2_e2e.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

29.3.2 MoCoV2

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc0712/mocoV2/rn50_
↪transfer_voc0712_detectron2_e2e.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

29.3.3 PIRL and NPID

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc0712/pirl_npid/rn50_
↪transfer_voc0712_npid_pirl.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

29.3.4 SimCLR and SwAV

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/voc0712/simclr_swav/
↪rn50_transfer_voc0712_simclr_swav.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

29.4 Benchmark: Mask R-CNN on COCO

Benchmarking on COCO introduces many variants (model architecture, FPN or not, C4). We provide config files for all the variants here and encourage users to pick the settings most suitable for their needs.

Benchmarking on COCO is not as widely adopted (compared to VOC07 and voc0712 evaluation) in self-supervision literature. This benchmark has been demonstrated extensively in MoCoV2 paper and we encourage users to refer to the paper.

An example run:

```
python tools/object_detection_benchmark.py \
    --config-file ../configs/config/benchmark/object_detection/COCOInstance/
↪sbnExtraNorm_precBN_r50_c4_coco.yaml \
    --num-gpus 8 MODEL.WEIGHTS <converted_d2_model_path>.torch
```

CHAPTER

THIRTY

HOW TO EXTRACT FEATURES

Given a pre-trained models, VISSL makes it easy to extract the features for the model on the datasets. VISSL seamlessly supports TorchVision models and in general, to load non-VISSL models, please follow our documentation for loading models.

To extract the features for a model that VISSL can load, users need 2 things:

- **config file**: the configuration file should clearly specify what layers of the model should features be extracted from.
- **set the correct engine_name**: in VISSL, we have two types of engine - a) training, b) feature extraction. Users must set `engine_name=extract_features` in the yaml config file.

Note: The SVM training and Nearest Neighbor benchmark workflows don't require setting the :code`engine_name` because the provided tools `train_svm` and `nearest_neighbor_test` explicitly add the feature extraction step.

30.1 Config File for Feature Extraction

Using the following examples, set the config options for your desired use case of feature extraction. Following examples are for ResNet-50 but users can use their model.

30.1.1 Extract features from several layers of the trunk

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["conv1", ["AvgPool2d", [[10, 10], 10, 4]]],
      ["res2", ["AvgPool2d", [[16, 16], 8, 0]]],
      ["res3", ["AvgPool2d", [[13, 13], 5, 0]]],
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
```

(continues on next page)

(continued from previous page)

```
      RESNETS:
        DEPTH: 50
```

30.1.2 Extract features of the trunk output

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
    SHOULD_FLATTEN_FEATS: False
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

30.1.3 Extract features of the model head output (self-supervised head)

For a given self-supervised approach, to extract the features of the model head (this is very common use case where the model head is a projection head and projects the trunk features into a low-dimensional space), The config settings should look like below. The example below is for SimCLR head + ResNet-50. Users can replace the `MODEL.HEAD.PARAMS` with the head settings used in the respective self-supervised model training.

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_AND_HEAD: True
    EVAL_TRUNK_AND_HEAD: True
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
```

30.2 How to extract features

Once users have the desired config file, user can extract features using the following command. VISSL also provides the config files here that users can modify/adapt to their needs.

```
python tools/run_distributed_engines.py \
    config=feature_extraction/extract_resnet_in1k_8gpu \
    +config/feature_extraction/trunk_only=rn50_layers \
    config.MODEL.WEIGHTS_INIT.PARAMS_FILE=<my_weights.torch>
```

CHAPTER

THIRTYONE

SUMMARY: FEATURE EVAL CONFIG SETTINGS

This doc describes summary of how to set :code:`MODEL.FEATURE_EVAL_SETTINGS` parameter for different evaluations.

In order to evaluate the model, you need to set :code:`MODEL.FEATURE_EVAL_SETTINGS` in yaml config file. Various options determine how the model is evaluated and also what part of the model is initialized from weights or what part of the model is frozen.

Below we provide instructions for setting the :code:`MODEL.FEATURE_EVAL_SETTINGS` for evaluating a pretrained model on several benchmark tasks. Below are only some example scenarios but hopefully provide an idea for any different use case one might have in mind.

31.1 Linear Image Classification with MLP heads

31.1.1 Attach MLP heads to several layers of the trunk

- If you want Zhang et. al style which has `BN -> FC` as the head, use `eval_mlp` head. Example:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["conv1", ["AvgPool2d", [[10, 10], 10, 4]]],
      ["res2", ["AvgPool2d", [[16, 16], 8, 0]]],
      ["res3", ["AvgPool2d", [[13, 13], 5, 0]]],
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
      ["eval_mlp", {"in_channels": 64, "dims": [9216, 1000]}],
      ["eval_mlp", {"in_channels": 256, "dims": [9216, 1000]}],
      ["eval_mlp", {"in_channels": 512, "dims": [8192, 1000]}],
      ["eval_mlp", {"in_channels": 1024, "dims": [9216, 1000]}],
```

(continues on next page)

(continued from previous page)

```
      ["eval_mlp", {"in_channels": 2048, "dims": [8192, 1000]}],
      ["eval_mlp", {"in_channels": 2048, "dims": [2048, 1000]}],
    ]
  WEIGHTS_INIT:
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

- If you want `FC` layer only in the head, use `mlp` head. Example:

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
        ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
        ["mlp", {"dims": [2048, 1000]}],
    ]
  WEIGHTS_INIT:
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

31.1.2 Attach MLP head to the trunk output

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    SHOULD_FLATTEN_FEATS: False
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
      ["eval_mlp", {"in_channels": 2048, "dims": [2048, 1000]}],
    ]
  WEIGHTS_INIT:
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

31.2 Linear Image Classification with SVM trainings

31.2.1 Train SVM on several layers of the trunk

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True    # only extract the features and we will train␣
↪SVM on these
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

31.2.2 Train SVM on the trunk output

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

31.3 Nearest Neighbor

31.3.1 knn test on trunk output

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True  # only freeze the trunk
    EXTRACT_TRUNK_FEATURES_ONLY: True    # we extract features from the trunk only
    SHOULD_FLATTEN_FEATS: False    # don't flatten the features and return as is
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  WEIGHTS_INIT:
```

(continues on next page)

(continued from previous page)

```
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

31.3.2 knn test on model head output (self-supervised head)

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_AND_HEAD: True   # both head and trunk will be frozen (including BN
↪in eval mode)
    EVAL_TRUNK_AND_HEAD: True  # initialized the model head as well from weights
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  HEAD:
    # SimCLR model head structure
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
  WEIGHTS_INIT:
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

31.3.3 knn test on several layers of the trunk

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True  # only freeze the trunk
    EXTRACT_TRUNK_FEATURES_ONLY: True   # we extract features from the trunk only
    SHOULD_FLATTEN_FEATS: False   # don't flatten the features and return as is
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
  WEIGHTS_INIT:
    PARAMS_FILE: ""
    STATE_DICT_KEY_NAME: classy_state_dict
```

31.4 Feature Extraction

You need to set `engine_name: extract_features` in the config file or pass the `engine_name=extract_features` as an additional input from the command line.

31.4.1 Extract features from several layers of the trunk

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
    SHOULD_FLATTEN_FEATS: False
    LINEAR_EVAL_FEAT_POOL_OPS_MAP: [
      ["conv1", ["AvgPool2d", [[10, 10], 10, 4]]],
      ["res2", ["AvgPool2d", [[16, 16], 8, 0]]],
      ["res3", ["AvgPool2d", [[13, 13], 5, 0]]],
      ["res4", ["AvgPool2d", [[8, 8], 3, 0]]],
      ["res5", ["AvgPool2d", [[6, 6], 1, 0]]],
      ["res5avg", ["Identity", []]],
    ]
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

31.4.2 Extract features of the trunk output

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_ONLY: True
    EXTRACT_TRUNK_FEATURES_ONLY: True
    SHOULD_FLATTEN_FEATS: False
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
```

31.4.3 Extract features of the model head output (self-supervised head)

```
MODEL:
  FEATURE_EVAL_SETTINGS:
    EVAL_MODE_ON: True
    FREEZE_TRUNK_AND_HEAD: True
    EVAL_TRUNK_AND_HEAD: True
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
```

(continues on next page)

(continued from previous page)

```
      RESNETS:
        DEPTH: 50
  HEAD:
    PARAMS: [
      ["mlp", {"dims": [2048, 2048], "use_relu": True}],
      ["mlp", {"dims": [2048, 128]}],
    ]
```

31.5 Full finetuning

Since this only requires to initialize the model from the pre-trained model weights, there's no need for the `FEATURE_EVAL_SETTINGS` params. Simply set `MODEL.WEIGHTS_INIT` params.

CHAPTER

THIRTYTWO

HOW TO LOAD PRETRAINED MODELS

VISSL supports Torchvision models out of the box. Generally, for loading any non-VISSL model, one needs to correctly set the following configuration options:

```
WEIGHTS_INIT:
  # path to the .torch weights files
  PARAMS_FILE: ""
  # name of the state dict. checkpoint = {"classy_state_dict": {layername:value}}.␣
↪Options:
  #   1. classy_state_dict - if model is trained and checkpointed with VISSL.
  #      checkpoint = {"classy_state_dict": {layername:value}}
  #   2. "" - if the model_file is not a nested dictionary for model weights i.e.
  #      checkpoint = {layername:value}
  #   3. key name that your model checkpoint uses for state_dict key name.
  #      checkpoint = {"your_key_name": {layername:value}}
  STATE_DICT_KEY_NAME: "classy_state_dict"
  # specify what layer should not be loaded. Layer names with this key are not copied
  # By default, set to BatchNorm stats "num_batches_tracked" to be skipped.
  SKIP_LAYERS: ["num_batches_tracked"]
  ####### If loading a non-VISSL trained model, set the following two args carefully #
↪########
  # to make the checkpoint compatible with VISSL, if you need to remove some names
  # from the checkpoint keys, specify the name
  REMOVE_PREFIX: ""
  # In order to load the model (if not trained with VISSL) with VISSL, there are 2␣
↪scenarios:
  #    1. If you are interested in evaluating the model features and freeze the trunk.
  #       Set APPEND_PREFIX="trunk.base_model." This assumes that your model is␣
↪compatible
  #       with the VISSL trunks. The VISSL trunks start with "_feature_blocks."␣
↪prefix. If
  #       your model doesn't have these prefix you can append them. For example:
  #       For TorchVision ResNet trunk, set APPEND_PREFIX="trunk.base_model._feature_
↪blocks."
  #    2. where you want to load the model simply and finetune the full model.
  #       Set APPEND_PREFIX="trunk."
  #       This assumes that your model is compatible with the VISSL trunks. The VISSL
  #       trunks start with "_feature_blocks." prefix. If your model doesn't have␣
↪these
  #       prefix you can append them.
  #       For TorchVision ResNet trunk, set APPEND_PREFIX="trunk._feature_blocks."
  # NOTE: the prefix is appended to all the layers in the model
  APPEND_PREFIX: ""
```

CHAPTER

THIRTYTHREE

TRAINING

The training in VISSL is composed of following components: Trainer, train task and train step

33.1 Trainer

The main entry point for any training or feature extraction workflows in VISSL if the trainer. It performs following:

- The trainer constructs a `train_task` which prepares all the components of the training (optimizer, loss, meters, model etc) using the settings specified by user in the yaml config file. Read below for details about train task.
- Setup the distributed training. VISSL support both GPU and CPU only training.
  - (1) Initialize the `torch.distributed.init_process_group` if the distributed is not already initialized. The init_method, backend are specified by user in the yaml config file. See VISSL defaults.yaml file for description on how to set `init_method`, `backend`.
  - (2) We also set the global cuda device index using torch.cuda.set_device or cpu device
- Executed the training or feature extraction workflows depending on :code`engine_name` set by users.

33.1.1 Training workflow

The training workflows executes the following steps. We get the training loop to use (vissl default is `standard_train_step` but the user can create their own training loop and specify the name `TRAINER.TRAIN_STEP_NAME`). The training happens:

1. Execute any hooks at the start of training (mostly resets the variable like iteration num phase_num etc)
2. For each epoch (train or test), run the hooks at the start of an epoch. Mostly involves setting things like timer, setting dataloader epoch etc
3. Execute the training loop (1 training iteration) involving forward, loss, backward, optimizer update, metrics collection etc.
4. At the end of epoch, sync meters and execute hooks at the end of phase. Involves things like checkpointing model, logging timers, logging to tensorboard etc

33.1.2 Feature extraction workflow

Set `engine_name=extract_features` in the config file to enable feature extraction.

Extract workflow supports multi-gpu feature extraction. Since we are only extracting features, only the model is built (and initialized from some model weights file if specified by user). The model is set to the eval mode fully. The features are extracted for whatever data splits (train, val, test) etc that user wants.

33.2 Train Task

A task prepares and holds all the components of a training like optimizer, datasets, dataloaders, losses, meters etc. Task also contains the variable like training iteration, epoch number etc. that are updated during the training.

We prepare every single component according to the parameter settings user wants and specified in the yaml config file.

Task also supports 2 additional things:

- converts the model BatchNorm layers to the synchronized batchnorm. Set the `MODEL.SYNC_BN_CONFIG.CONVERT_BN_TO_SYNC_BN=true`
- sets mixed precision (apex and pytorch both supported). Set the `MODEL.AMP_PARAMS.USE_AMP=true` and select the desired AMP settings.

33.3 Train Loop

VISSL implements a default training loop (single iteration step) that is used for self-supervised training of all VISSL reference approaches, for feature extraction and for supervised workflows. Users con implement their own training loop.

The training loop performs: data read, forward, loss computation, backward, optimizer step, parameter updates.

Various intermediate steps are also performed:

- logging the training loss, training eta, LR, etc to loggers
- logging to tensorboard,
- performing any self-supervised method specific operations (like in MoCo approach, the momentum encoder is updated), computing the scores in swav
- checkpointing model if user wants to checkpoint in the middle of an epoch

To select the training loop:

```
TRAINER:
  # default training loop. User can define their own loop and use that instead.
 TRAIN_STEP_NAME: "standard_train_step"
```

CHAPTER

THIRTYFOUR

BUILDING MODELS

The model in VISSL is split into `trunk` that computes features and `head` that computes outputs (projections, classifications etc).

VISSL supports several types of Heads and several types of trunks. Overall, the following use cases are supported by VISSL models:

- Model producing single output as in standard supervised ImageNet training
- Model producing multiple outputs (Multi-task)
- Model producing multiple outputs from different features (layers) from the trunk (useful in linear evaluation of features from several model layers)
- Model that accepts multiple inputs (e.g. image and patches as in PIRL appraoch).
- Model where the trunk is frozen and head is trained
- Model that supports multiple resolutions inputs as in SwAV
- Model that is completely frozen and features are extracted.

34.1 Trunks

VISSL supports many trunks including AlexNet (variants for approaches like Jigsaw, Colorization, RotNet, DeepCluster etc), ResNets, ResNeXt, RegNets, EfficientNet.

To set the trunk, user needs to specify the trunk name in `MODEL.TRUNK.NAME`.

Examples of trunks:

- **Using ResNe(X)ts trunk:**

```
MODEL:
  TRUNK:
    NAME: resnet
    TRUNK_PARAMS:
      RESNETS:
        DEPTH: 50
        WIDTH_MULTIPLIER: 1
        NORM: BatchNorm    # BatchNorm | LayerNorm
        GROUPS: 1
        ZERO_INIT_RESIDUAL: False
        WIDTH_PER_GROUP: 64
        # Colorization model uses stride=1 for last layer to retain higher spatial␣
↪resolution
```

(continues on next page)

(continued from previous page)

```
        # for the pixel-wise task. Torchvision default is stride=2 and all other␣
↪models
        # use this so we set the default as 2.
        LAYER4_STRIDE: 2
```

- **Using RegNets trunk**: We follow RegNets defined in ClassyVision directly and users can either use a pre-defined ClassyVision RegNet config or define their own.
  - for example, to create a new RegNet config for RegNet-256Gf model (1.3B params):

```
MODEL:
  TRUNK:
    NAME: regnet
    TRUNK_PARAMS:
      REGNET:
        depth: 27
        w_0: 640
        w_a: 230.83
        w_m: 2.53
        group_width: 373
```

  - To use a pre-defined RegNet config in classy vision example: RegNetY-16gf

```
MODEL:
  TRUNK:
    NAME: regnet_y_16gf
```

34.2 Heads

This function creates the heads needed by the module. The head is specified by setting `MODEL.HEAD.PARAMS` in the configuration file.

The `MODEL.HEAD.PARAMS` is a list of Pairs containing parameters for (multiple) heads.

- Pair[0] = Name of Head.
- Pair[1] = kwargs passed to head constructor.

Example of ["name", kwargs] `MODEL.HEAD.PARAMS=["mlp", {"dims": [2048, 128]}]`

34.2.1 Types of Heads one can specify

- **Case1: Simple Head containing single module - Single Input, Single output**

```
MODEL:
  HEAD:
    PARAMS: [
        ["mlp", {"dims": [2048, 128]}]
    ]
```

- **Case2: Complex Head containing chain of head modules - Single Input, Single output**

```
MODEL:
  HEAD:
    PARAMS: [
         ["mlp", {"dims": [2048, 1000], "use_bn": False, "use_relu": False}],
         ["siamese_concat_view", {"num_towers": 9}],
         ["mlp", {"dims": [9000, 128]}]
    ]
```

- **Case3: Multiple Heads (example 2 heads) - Single input, multiple output**: can be used for multi-task learning

```
MODEL:
  HEAD:
    PARAMS: [
         # head 0
         [
              ["mlp", {"dims": [2048, 128]}]
         ],
         # head 1
         [
              ["mlp", {"dims": [2048, 1000], "use_bn": False, "use_relu": False}],
              ["siamese_concat_view", {"num_towers": 9}],
              ["mlp", {"dims": [9000, 128]}],
         ]
    ]
```

- **Case4: Multiple Heads (example 5 simple heads) - Single input, multiple output:**: For example, used in linear evaluation of models

```
MODEL:
  HEAD:
    PARAMS: [
         ["eval_mlp", {"in_channels": 64, "dims": [9216, 1000]}],
         ["eval_mlp", {"in_channels": 256, "dims": [9216, 1000]}],
         ["eval_mlp", {"in_channels": 512, "dims": [8192, 1000]}],
         ["eval_mlp", {"in_channels": 1024, "dims": [9216, 1000]}],
         ["eval_mlp", {"in_channels": 2048, "dims": [8192, 1000]}],
    ]
```

34.2.2 Applying heads on multiple trunk features

By default, the head operates on the trunk output (single or multiple output). However, one can explicitly specify the `input` to heads mapping in the list `MODEL.MULTI_INPUT_HEAD_MAPPING`. This is used in PIRL training.

Assumptions:

- This assumes that the same trunk is used to extract features for the different types of inputs.
- One head only operates on one kind of input, Every individual head can contain several layers as in Case2 above.

`MODEL.MULTI_INPUT_HEAD_MAPPING` specifies Input -> Trunk Features mapping. Like in the single input case, the heads can operate on features from different layers. In this case, we specify `MODEL.MULTI_INPUT_HEAD_MAPPING` to be a list like:

```
MODEL:
  MULTI_INPUT_HEAD_MAPPING: [
         ["input_key", [list of features heads is applied on]]
  ]
```

For example: for a model that applies two heads on images and one head on patches:

```
MODEL:
    MULTI_INPUT_HEAD_MAPPING: [
        ["images", ["res5", "res4"]],
        ["patches", ["res3"]
    ],
```

CHAPTER

THIRTYFIVE

USING OPTIMIZERS

VISSL support all PyTorch optimizers (SGD, Adam etc) and ClassyVision optimizers.

35.1 Creating Optimizers

The optimizers can be easily created from the configuration files. The user needs to set the optimizer name in `OPTIMIZER.name`. Users can configure other settings like #epochs, etc as follows:

```
OPTIMIZER:
    name: sgd
    weight_decay: 0.0001
    momentum: 0.9
    nesterov: False
    # for how many epochs to do training. only counts training epochs.
    num_epochs: 90
    # whether to regularize batch norm. if set to False, weight decay of batch norm␣
↪params is 0.
    regularize_bn: False
    # whether to regularize bias parameter. if set to False, weight decay of bias␣
↪params is 0.
    regularize_bias: True
```

35.1.1 Using different LR for Head and trunk

VISSL supports using a different LR and weight decay for head and trunk. User needs to set the config option `OPTIMIZER.head_optimizer_params.use_different_values=True` in order to enable this functionality.

```
OPTIMIZER:
  head_optimizer_params:
    # if the head should use a different LR than the trunk. If yes, then specify the
    # param_schedulers.lr_head settings. Otherwise if set to False, the
    # param_scheduelrs.lr will be used automatically.
    use_different_lr: False
    # if the head should use a different weight decay value than the trunk.
    use_different_wd: False
    # if using different weight decay value for the head, set here. otherwise, the
    # same value as trunk will be automatically used.
    weight_decay: 0.0001
```

35.1.2 Using LARC

VISSL supports the LARC implementation from NVIDIA's Apex LARC. To use LARC, users need to set config option `OPTIMIZER.use_larc=True`. VISSL exposes LARC parameters that users can tune. Full list of LARC parameters exposed by VISSL:

```
OPTIMIZER:
  name: "sgd"
  use_larc: False  # supported for SGD only for now
  larc_config:
    clip: False
    eps: 1e-08
    trust_coefficient: 0.001
```

Note: LARC is currently supported for SGD optimizer only.

35.2 Creating LR Schedulers

Users can use different types of Learning rate schedules for the training of their models. We closely follow the LR schedulers supported by ClassyVision and also custom learning rate schedules in VISSL.

35.2.1 How to set learning rate

Below we provide some examples of how to setup various types of Learning rate schedules. Note that these are merely some examples and you should set your desired parameter values.

1. Cosine

```
OPTIMIZER:
  param_schedulers:
    lr:
      name: cosine
      start_value: 0.15    # LR for batch size 256
      end_value: 0.0000
```

2. Multi-Step

```
OPTIMIZER:
  param_schedulers:
    lr:
      name: multistep
      values: [0.01, 0.001]
      milestones: [1]
      update_interval: epoch  # update LR after every epoch
```

3. Linear Warmup + Cosine

```
OPTIMIZER:
  param_schedulers:
    lr:
      name: composite
      schedulers:
```

(continues on next page)

(continued from previous page)

```
        - name: linear
            start_value: 0.6
            end_value: 4.8
        - name: cosine
            start_value: 4.8
            end_value: 0.0048
      interval_scaling: [rescaled, fixed]
      update_interval: step
      lengths: [0.1, 0.9]                 # 100ep
```

4. Cosine with restarts

```
OPTIMIZER:
  param_schedulers:
    lr:
      name: cosine_warm_restart
      start_value: 0.15    # LR for batch size 256
      end_value: 0.00015
      restart_interval_length: 0.5
      wave_type: half  # full | half
```

5. Linear warmup + cosine with restarts

```
OPTIMIZER:
  param_schedulers:
    lr:
      name: composite
      schedulers:
        - name: linear
            start_value: 0.6
            end_value: 4.8
        - name: cosine_warm_restart
            start_value: 4.8
            end_value: 0.0048
            # wave_type: half
            # restart_interval_length: 0.5
            wave_type: full
            restart_interval_length: 0.334
      interval_scaling: [rescaled, rescaled]
      update_interval: step
      lengths: [0.1, 0.9]                 # 100ep
```

6. Multiple linear warmups and cosine

```
OPTIMIZER:
  param_schedulers:
    lr:
      schedulers:
        - name: linear
            start_value: 0.6
            end_value: 4.8
        - name: cosine
            start_value: 4.8
            end_value: 0.0048
        - name: linear
            start_value: 0.0048
            end_value: 2.114
```

(continues on next page)

(continued from previous page)

```
        - name: cosine
            start_value: 2.114
            end_value: 0.0048
      update_interval: step
      interval_scaling: [rescaled, rescaled, rescaled, rescaled]
      lengths: [0.0256, 0.48722, 0.0256, 0.46166]          # 1ep IG-500M
```

35.2.2 Auto-scaling of Learning Rate

VISSL supports automatically scaling LR as per https://arxiv.org/abs/1706.02677. To turn this automatic scaling on, set `config.OPTIMIZER.param_schedulers.lr.auto_lr_scaling.auto_scale=true`.

`scaled_lr` is calculated: for a given

- `base_lr_batch_size` = batch size for which the base learning rate is specified,
- `base_value` = base learning rate value that will be scaled, the current batch size is used to determine how to scale the base learning rate value.

```
scaled_lr = ((batchsize_per_gpu * world_size) * base_value ) /
base_lr_batch_size
```

For different types of learning rate schedules, the LR scaling is handles as below:

```
1. cosine:
    end_value = scaled_lr * (end_value / start_value)
    start_value = scaled_lr and
2. multistep:
    gamma = values[1] / values[0]
    values = [scaled_lr * pow(gamma, idx) for idx in range(len(values))]
3. step_with_fixed_gamma
    base_value = scaled_lr
4. linear:
   end_value = scaled_lr
5. inverse_sqrt:
   start_value = scaled_lr
6. constant:
   value = scaled_lr
7. composite:
    recursively call to scale each composition. If the composition consists of a␣
↪linear
    schedule, we assume that a linear warmup is applied. If the linear warmup is
    applied, it's possible the warmup is not necessary if the global batch_size is␣
↪smaller
    than the base_lr_batch_size and in that case, we remove the linear warmup from the
    schedule.
```

CHAPTER

THIRTYSIX

USING PYTORCH AND VISSL LOSSES

VISSL supports all PyTorch loss functions and also implements several loss functions that are specific to self-supervised approaches like MoCo, PIRL, SwAV, SimCLR etc. Using any loss is very easy in VISSL and involves simply editing the configuration files to specify the loss name and the parameters of that loss. See all the VISSL custom losses here.

To use a certain loss, users need to simply set `LOSS.name=<my_loss_name>` and set the parameter values that loss requires.

Examples:

- Using Cross entropy loss for training and testing

```
LOSS:
  name: "CrossEntropyLoss"
  # ------------------------------------------------------------------------------------
↪- #
  # Standard PyTorch Cross-Entropy Loss. Use the loss name exactly as in PyTorch.
  # pass any variables that the loss takes.
  # ------------------------------------------------------------------------------------
↪- #
  CrossEntropyLoss:
    ignore_index: -1
```

- Using SwAV loss for training, sim

```
LOSS:
  name: swav_loss
  swav_loss:
    temperature: 0.1
    use_double_precision: False
    normalize_last_layer: True
    num_iters: 3
    epsilon: 0.05
    crops_for_assign: [0, 1]
    temp_hard_assignment_iters: 0
    num_crops: 2                    # automatically inferred from data transforms
    num_prototypes: [3000]          # automatically inferred from model HEAD settings
    embedding_dim: 128              # automatically inferred from HEAD params
    # for dumping the debugging info in case loss becomes NaN
    output_dir: ""                  # automatically inferred and set to checkpoint dir
    queue:
      local_queue_length: 0         # automatically inferred to queue_length // world_
↪size
      queue_length: 0               # automatically adjusted to ensure queue_length %␣
↪global batch size = 0
```

(continues on next page)

(continued from previous page)

```
    start_iter: 0
```

CHAPTER

THIRTYSEVEN

USING METERS

VISSL supports PyTorch meters and implements some custom meters that like Mean Average Precision meter. Meters in VISSL support single target multiple outputs. This is especially useful and relvant during evaluation of self-supervised models where we want to calculate feature quality of several layers of the model. See all the VISSL custom meters here.

To use a certain meter, users need to simply set `METERS.name=<my_meter_name>` and set the parameter values that meter requires.

Examples:

- Using Accuracy meter to compute Top-k accuracy for training and testing

```
METERS:
  name: accuracy_list_meter
  accuracy_list_meter:
    num_meters: 1           # number of outputs model has. also auto inferred
    topk_values: [1, 5]     # for each meter, what topk are computed.
```

- Using Mean AP meter:

```
METERS:
  name: mean_ap_list_meter
  mean_ap_list_meter:
    num_classes: 9605   # openimages v6 dataset classes
    num_meters: 1
```

CHAPTER

THIRTYEIGHT

HOOKS

Hooks are the helper functions that can be executed at several parts of a training process as described below:

- `on_start`: These hooks are executed before the training starts.
- `on_phase_start`: executed at the beginning of every epoch (including test, train epochs)
- `on_forward`: executed after every forward pass
- `on_loss_and_meter`: executed after loss and meters are calculateds
- `on_backward`: executed after every backward pass of the model
- `on_update`: executed after model parameters are updated by the optimizer
- `on_step`: executed after one single training (or test) iteration finishes
- `on_phase_end`: executed after the epoch (train or test) finishes
- `on_end`: executed at the very end of training.

Hooks are executed by inserting `task.run_hooks(SSLClassyHookFunctions.<type>.name)` at several steps of the training.

38.1 How to enable certain hooks in VISSL

VISSL supports many hooks. Users can configure which hooks to use from simple configuration files. The hooks in VISSL can be categorized into following buckets:

- `Tensorboard hook`: to enable this hook, set `TENSORBOARD_SETUP.USE_TENSORBOARD=true` and configure the tensorboard settings
- `Model Complexity hook`: this hook performs one single forward pass of the model on the synthetic input and computes the #FLOPs, #params and #activations in the model. To enable this hook, set `MODEL.MODEL_COMPLEXITY.COMPUTE_COMPLEXITY=true` and configure it.
- `Self-supervised Loss hooks`: VISSL has hooks specific to self-supervised approaches like MoCo, SwAV etc. These hooks are handy in performing some intermediate operations required in self-supervision. For example: `MoCoHook` is called after every forward pass of the model and updates the momentum encoder network. Users don't need to do anything special for using these hooks. If the user configuration file has the loss function for an approach, VISSL will automatically enable the hooks for the approach.
- `Logging, checkpoint, training variable update hooks`: These hooks are used by default in VISSL and perform operations like logging the training progress (loss, LR, eta etc) on stdout, save checkpoints etc.

CHAPTER

THIRTYNINE

USING DATA

To use a dataset in VISSL, the only requirements are:

- the dataset name should be registered with `VisslDatasetCatalog` in VISSL. Only name is important and the paths are not. The paths can be specifed in the configuration file. Users can either edit the `dataset_catalog.json` or specify the paths in the configuration file.
- the dataset should be from a supported data source.

39.1 Reading data from several sources

VISSL allows reading data from multiple sources (disk, etc) and in multiple formats (a folder path, a `.npy` file). The GenericSSLDataset class is defined to support reading data from multiple data sources. For example: `data = [dataset1, dataset2]` and the minibatches generated will have the corresponding data from each dataset. For this reason, we also support labels from multiple sources. For example `targets = [dataset1 targets, dataset2 targets]`.

Source of the data (`disk_filelist | disk_folder`):

- `disk_folder`: this is simply the root folder path to the downloaded data.
- `disk_filelist`: These are numpy (or .pkl) files: (1) file containing images information (2) file containing corresponding labels for images. We provide scripts that can be used to prepare these two files for a dataset of choice.

To use a dataset, VISSL takes following inputs in the configuration file for each dataset split (train, test):

- `DATASET_NAMES`: names of the datasets that are registered with `VisslDatasetCatalog`. Registering dataset name is important. Example: `DATASET_NAMES=[imagenet1k_folder, my_new_dataset_filelist]`
- `DATA_SOURCES`: the sources of dataset. Options: `disk_folder | disk_filelist`. This specifies where the data lives. Users can extend it for their purposes. Example `DATA_SOURCES=[disk_folder, disk_filelist]`
- `DATA_PATHS`: the paths to the dataset. The paths could be folder path (example Imagenet1k folder) or .npy filepaths. For the folder paths, VISSL uses `ImageFolder` from PyTorch. Example `DATA_PATHS=[<imagenet1k_folder_path>, <numpy_file_path_for_new_dataset>]`
- `LABEL_SOURCES`: just like images, the targets can also come from several sources. Example: `LABEL_SOURCES=[disk_folder]` for Imagenet1k. Example: `DATA_SOURCES=[disk_folder, disk_filelist]`
- `LABEL_PATHS`: similar to `DATA_PATHS` but for labels. Example `LABEL_PATHS=[<imagenet1k_folder_path>, <numpy_file_path_for_new_dataset_labels>]`

- LABEL_TYPE: choose from standard | sample_index. sample_index is a common practice in self-supervised learning and sample_index`=id of the sample in the data. :code:`standard label type is used for supervised learning and user specifis the annotated labels to use.

39.2 Using dataset_catalog.json

In order to use a dataset with VISSL, the dataset name must be registered with VisslDatasetCatalog. VISSL maintains a dataset_catalog.json which is parsed by VisslDatasetCatalog and the datasets are registered with VISSL, ready-to-use.

Users can edit the template dataset_catalog.json file to specify their datasets paths. The json file can be fully decided by user and can have any number of supported datasets (one or more). User can give the string names to dataset as per their choice.

39.2.1 Template for a dataset entry in dataset_catalog.json

```
"data_name": {
   "train": [
     "<images_path_or_folder>", "<labels_path_or_folder>"
   ],
   "val": [
     "<images_path_or_folder>", "<labels_path_or_folder>"
   ],
}
```

The images_path_or_folder and labels_path_or_folder can be directories or filepaths (numpy, pickle.)

User can mix match the source of image, labels i.e. labels can be filelist and images can be folder path. The yaml configuration files require specifying LABEL_SOURCES and DATA_SOURCES which allows the code to figure out how to ingest various sources.

Note: Filling the dataset_catalog.json is a one time process only and provides the benefits of simply accessing any dataset with the dataset name in the configuration files for the rest of the trainings.

39.3 Using Builtin datasets

VISSL supports several Builtin datasets as indicated in the dataset_catalog.json file. Users can specify paths to those datasets.

39.3.1 Expected dataset structure for ImageNet, Places205, Places365

```
{imagenet, places205, places365}
train/
    <n0......>/
        <im-1-name>.JPEG
        ...
        <im-N-name>.JPEG
        ...
    <n1......>/
        <im-1-name>.JPEG
        ...
        <im-M-name>.JPEG
        ...
    ...
val/
    <n0......>/
        <im-1-name>.JPEG
        ...
        <im-N-name>.JPEG
        ...
    <n1......>/
        <im-1-name>.JPEG
        ...
        <im-M-name>.JPEG
        ...
    ...
```

39.3.2 Expected dataset structure for Pascal VOC [2007, 2012]

```
VOC20{07,12}/
    Annotations/
    ImageSets/
        Main/
        trainval.txt
        test.txt
    JPEGImages/
```

39.3.3 Expected dataset structure for COCO2014

```
coco/
    annotations/
        instances_train2014.json
        instances_val2014.json
    train2014/
        # image files that are mentioned in the corresponding json
    val2014/
        # image files that are mentioned in the corresponding json
```

39.4 Dataloader

VISSL uses PyTorch `torch.utils.data.DataLoader` and allows setting all the dataloader option as below. The dataloader is wrapped with DataloaderAsyncGPUWrapper or DataloaderSyncGPUWrapper depending on whether user wants to copy data to gpu async or not.

The settings for the `Dataloader` in VISSL are:

```
dataset (GenericSSLDataset):    the dataset object for which dataloader is constructed
dataset_config (dict):          configuration of the dataset. it should be DATA.TRAIN␣
↪or DATA.TEST settings
num_dataloader_workers (int):   number of workers per gpu (or cpu) training
pin_memory (bool):              whether to pin memory or not
multi_processing_method (str):  method to use. options: forkserver | fork | spawn
device (torch.device):          training on cuda or cpu
get_sampler (get_sampler):      function that is used to get the sampler
worker_init_fn (None default):  any function that should be executed during␣
↪initialization of dataloader workers
```

39.5 Using Data Collators

VISSL supports PyTorch default collator `torch.utils.data.dataloader.default_collate` and also many custom data collators used in self-supervision. The use any collator, user has to simply specify the `DATA.TRAIN.COLLATE_FUNCTION` to be the name of the collator to use. See all custom VISSL collators implemented here.

An example for specifying collator for SwAV training:

```
DATA:
  TRAIN:
    COLLATE_FUNCTION: multicrop_collator
```

39.6 Using Data Transforms

VISSL supports all PyTorch `TorchVision` transforms as well as many transforms required by Self-supervised approaches including MoCo, SwAV, PIRL, SimCLR, BYOL, etc. Using Transforms is very intuitive and easy in VISSL. Users specify the list of transforms they want to apply on the data in the order of application. This involves using the transform name and the key:value to specify the parameter values for the transform. See the full list of transforms implemented by VISSL here

An example of transform for SwAV:

```
DATA:
  TRAIN:
    TRANSFORMS:
      - name: ImgPilToMultiCrop
        total_num_crops: 6
        size_crops: [224, 96]
        num_crops: [2, 4]
        crop_scales: [[0.14, 1], [0.05, 0.14]]
      - name: RandomHorizontalFlip
        p: 0.5
```

(continues on next page)

(continued from previous page)

```
    - name: ImgPilColorDistortion
      strength: 1.0
    - name: ImgPilGaussianBlur
      p: 0.5
      radius_min: 0.1
      radius_max: 2.0
    - name: ToTensor
    - name: Normalize
      mean: [0.485, 0.456, 0.406]
      std: [0.229, 0.224, 0.225]
```

39.7 Using Data Sampler

VISSL supports 2 types of samplers:

- PyTorch default `torch.utils.data.distributed.DistributedSampler`
- VISSL sampler StatefulDistributedSampler that is written specifically for large scale dataset trainings. See the documentation for the sampler.

By default, the PyTorch default sampler is used unless user specifies `DATA.TRAIN.USE_STATEFUL_DISTRIBUTED_SAMPLER=true` in which case `StatefulDistributedSampler` will be used.

CHAPTER

FORTY

ADD CUSTOM TRAIN LOOP

VISSL implements a default training loop (single iteration step) that is used for self-supervised training of all VISSL reference approaches, for feature extraction and for supervised workflows. Users con implement their own training loop.

The training loop performs: data read, forward, loss computation, backward, optimizer step, parameter updates.

Various intermediate steps are also performed:

- logging the training loss, training eta, LR, etc to loggers
- logging to tensorboard,
- performing any self-supervised method specific operations (like in MoCo approach, the momentum encoder is updated), computing the scores in swav
- checkpointing model if user wants to checkpoint in the middle of an epoch

Users can implement their custom training loop by following the steps:

- **Step1**: Create your `my_new_training_loop` module under `vissl/trainer/train_steps/my_new_training_loop.py` following the template:

```
from vissl.trainer.train_steps import register_train_step

@register_train_step("my_new_training_loop")
def my_new_training_loop(task):
    """
    add documentation on what this training loop does and how it varies from
    standard training loop in vissl.
    """
    # implement the training loop. It should take care of running the dataloader
    # iterator to get the input sample
    ...
    ...

    return task
```

- **Step2**: New train loop is ready to use. Set the `TRAINER.TRAIN_STEP_NAME=my_new_training_loop`

CHAPTER

FORTYONE

ADD NEW HOOKS

Hooks are the helper functions that can be executed at several parts of a training process as described below:

- `on_start`: These hooks are executed before the training starts.
- `on_phase_start`: executed at the beginning of every epoch (including test, train epochs)
- `on_forward`: executed after every forward pass
- `on_loss_and_meter`: executed after loss and meters are calculateds
- `on_backward`: executed after every backward pass of the model
- `on_update`: executed after model parameters are updated by the optimizer
- `on_step`: executed after one single training (or test) iteration finishes
- `on_phase_end`: executed after the epoch (train or test) finishes
- `on_end`: executed at the very end of training.

Hooks are executed by inserting `task.run_hooks(SSLClassyHookFunctions.<type>.name)` at several steps of the training.

Users can add new hooks by following the steps below:

- **Step1**: Create your new hook in `vissl/hooks/my_hook.py` following the template.

```
from classy_vision.hooks.classy_hook import ClassyHook

class MyAwesomeNewHook(ClassyHook):
    """
    Logs the number of paramaters, forward pass FLOPs and activations of the model.
    Adapted from: https://github.com/facebookresearch/ClassyVision/blob/master/classy_
↪vision/hooks/model_complexity_hook.py#L20    # NOQA
    """

    # define all the functions that your hook should execute. If the hook
    # executes nothing for a particular function, mark it as a noop.
    # Example: if the hook only does something for `on_start', then set:
    #    on_phase_start = ClassyHook._noop
    #    on_forward = ClassyHook._noop
    #    on_loss_and_meter = ClassyHook._noop
    #    on_backward = ClassyHook._noop
    #    on_update = ClassyHook._noop
    #    on_step = ClassyHook._noop
    #    on_phase_end = ClassyHook._noop
    #    on_end = ClassyHook._noop
```

(continues on next page)

(continued from previous page)

```
    def on_start(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute at the beginning of training
        ...

    def on_phase_start(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute at the beginning of each epoch
        # (training or test epoch). Use `task.train' boolean to detect if the current
        # epoch is train or test
        ...

    def on_forward(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after the model forward pass is done
        # should handle the train or test phase.
        # Use `task.train' boolean to detect if the current epoch is train or test
        ...

    def on_loss_and_meter(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after the loss and meters are
        # calculated
        ...

    def on_backward(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after backward pass is done. Note
        # that the model parameters are not yet updated
        ...

    def on_update(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after the model parameters are␣
↪updated
        # by the optimizer following LR and weight decay
        ...

    def on_step(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after a training / test iteration
        # is done
        ...

    def on_phase_end(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute after a phase (train or test)
        # is done
        ...

    def on_end(self, task: "tasks.ClassyTask") -> None:
        # implement what your hook should execute at the end of training
        # (or testing, feature extraction)
        ...
```

- **Step2**: Inform VISSL on how/when to use the hook in `default_hook_generator` method in `vissl/hooks/__init__.py`. We recommend adding some configuration params like `MONITOR_PERF_STATS` in `vissl/config/defaults.yaml` so that you can set the usage of hook easily from the config file.
- **Step3**: Test your hook is working by simply running a config file and setting the config parameters you added in Step2 above.

CHAPTER

FORTYTWO

ADD NEW OPTIMIZERS

VISSL makes it easy to add new optimizers. VISSL depends on ClassyVision .

Follow the steps below to add a new optimizer to VISSL.

- **Step1**: Create your new optimizer `my_new_optimizer` under `vissl/optimizers/my_new_optimizer.py` following the template:

```
from classy_vision.optim import ClassyOptimizer, register_optimizer

@register_optimizer("my_new_optimizer")
class MyNewOptimizer(ClassyOptimizer):
    """
    Add documentation on how the optimizer optimizes and also
    link to any papers or techincal reports that propose/use the
    the optimizer (if applicable)
    """
    def __init__(self, param1, param2, ...):
        super().__init__()
        # implement what the optimizer init should do and what variable it should␣
↪store
        ...


    def prepare(self, param_groups):
        """
        Prepares the optimizer for training.

        Deriving classes should initialize the underlying PyTorch
        :class:`torch.optim.Optimizer` in this call. The param_groups argument
        follows the same format supported by PyTorch (list of parameters, or
        list of param group dictionaries).

        Warning:
            This should called only after the model has been moved to the correct
            device (gpu or cpu).
        """

    @classmethod
    def from_config(cls, config: Dict[str, Any]) -> "SGD":
        """
        Instantiates a MyNewOptimizer from a configuration.

        Args:
            config: A configuration for a MyNewOptimizer.
                See :func:`__init__` for parameters expected in the config.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            A MyNewOptimizer instance.
        """
```

- **Step2**: Enable the automatic import of all the modules. Add the following lines of code to `vissl/optimizers/__init__.py`. Skip this step if already exists.

```
from pathlib import Path
from classy_vision.generic.registry_utils import import_all_modules

FILE_ROOT = Path(__file__).parent

# automatically import any Python files in the optimizers/ directory
import_all_modules(FILE_ROOT, "vissl.optimizers")
```

- **Step3**: Enable the registry of the new optimizers in VISSL. Add the following line to `vissl/trainer/__init__.py`. Skip this step if already exists.

```
import vissl.optimizers # NOQA
```

- **Step4**: The optimizer is now ready to use. Set the configuration param `OPTIMIZER.name=my_new_optimizer` and set the values of the params this optimizer takes.

CHAPTER

FORTYTHREE

ADD NEW LR SCHEDULERS

VISSL allows adding new Learning rate schedulers easily. Follow the steps below:

- **Step1**: Create a class for your `my_new_lr_scheduler` under `vissl/optimizers/param_scheduler/my_new_lr_scheduler.py` following the template:

```
from classy_vision.optim.param_scheduler import (
    ClassyParamScheduler,
    UpdateInterval,
    register_param_scheduler,
)

@register_param_scheduler("my_new_lr_scheduler")
class MyNewLRScheduler(ClassyParamScheduler):
    """
    Add documentation on how the LR schedulers works and also
    link to any papers or techincal reports that propose/use the
    the scheduler (if applicable)

    Args:
        document all the inputs that the scheduler takes

    Example:
        show one example of how to use the lr scheduler
    """

    def __init__(
        self, param1, param2, ... , update_interval: UpdateInterval = UpdateInterval.
↪STEP
    ):

        super().__init__(update_interval=update_interval)

        # implement what the init of LR scheduler should do, any variables
        # to initialize etc.
        ...
        ...

    @classmethod
    def from_config(cls, config: Dict[str, Any]) -> "MyNewLRScheduler":
        """
        Instantiates a MyNewLRScheduler from a configuration.

        Args:
            config: A configuration for a MyNewLRScheduler.
```

(continues on next page)

(continued from previous page)

```
            See :func:`__init__` for parameters expected in the config.

        Returns:
            A MyNewLRScheduler instance.
        """
        return cls(param1=config.param1, param2=config.param2, ...)

    def __call__(self, where: float):
        # implement what the LR value should be give the `where' which indicates
        # how far the training is. `where' values are [0, 1)
        ...
        ...

        return lr_value
```

- **Step2**: The new LR scheduler is ready to use. Give it a try by setting configuration param `OPTIMIZER.param_schedulers.lr.name=my_new_lr_scheduler`.

CHAPTER

FORTYFOUR

ADD NEW LOSSES TO VISSL

VISSL allows adding new losses easily. Follow the steps below to add a new loss:

- **Step1**: Create a new loss `my_new_loss` in `vissl/losses/my_new_loss.py` following the template

```
import pprint
from classy_vision.losses import ClassyLoss, register_loss

@register_loss("my_new_loss")
class MyNewLoss(ClassyLoss):
    """
    Add documentation for what the loss does

    Config params:
        document what parameters should be expected for the loss in the defaults.yaml
        and how to set those params
    """

    def __init__(self, loss_config: AttrDict, device: str = "gpu"):
        super(MyNewLoss, self).__init__()

        self.loss_config = loss_config
        # implement what the init function should do
        ...

    @classmethod
    def from_config(cls, loss_config: AttrDict):
        """
        Instantiates MyNewLoss from configuration.

        Args:
            loss_config: configuration for the loss

        Returns:
            MyNewLoss instance.
        """
        return cls(loss_config)

    def __repr__(self):
        # implement what information about loss params should be
        # printed by print(loss). This is helpful for debugging
        repr_dict = {"name": self._get_name(), ....}
        return pprint.pformat(repr_dict, indent=2)

    def forward(self, output, target):
```

(continues on next page)

(continued from previous page)

```
        # implement how the loss should be calculated. The output should be
        # torch.Tensor or List[torch.Tensor] and target should be torch.Tensor
        ...
        ...

        return loss
```

- **Step2**: Register the loss and loss params with VISSL Configuration. Add the params that the loss takes in VISSL defaults.yaml as follows:

```
LOSS:
  my_new_loss:
    param1: value1
    param2: value2
    ...
```

- **Step3**: Loss is ready to use. Simply set the configuration param `LOSS.name=my_new_loss`

CHAPTER

FORTYFIVE

ADD NEW METERS

VISSL allows adding new meters easily. Follow the steps below to add a new loss:

- **Step1**: Create a new loss `my_new_meter` in `vissl/meters/my_new_meter.py` following the template

```
from classy_vision.meters import ClassyMeter, register_meter

@register_meter("my_new_meter")
class MyNewMeter(ClassyMeter):
    """
    Add docuementation on what this meter does

    Args:
        add documentation about each meter parameter
    """

    def __init__(self, meters_config: AttrDict):
        # implement what the init method should do like
        # setting variable to update etc.
        self.reset()

    @classmethod
    def from_config(cls, meters_config: AttrDict):
        """
        Get the MyNewMeter instance from the user defined config
        """
        return cls(meters_config)

    @property
    def name(self):
        """
        Name of the meter
        """
        return "my_new_meter"

    @property
    def value(self):
        """
        Value of the meter which has been globally synced. This is the value printed␣
↪and
        recorded by user.
        """
        # implement how the value should be calculated/finalized/returned to user
        ....
```

(continues on next page)

(continued from previous page)

```
        return {"my_metric_name": value, ....}

    def sync_state(self):
        """
        Globally syncing the state of each meter across all the trainers.
        Should perform distributed communications like all_gather etc
        to correctly gather the global values to compute the metric
        """
        # implement what Communications should be done to globally sync the state
        ...

        # update the meter variables to store these global gathered values
        ...

    def reset(self):
        """
        Reset the meter. Should reset all the meter variables, values.
        """
        self._scores = torch.zeros(0, self.num_classes, dtype=torch.float32)
        self._targets = torch.zeros(0, self.num_classes, dtype=torch.int8)
        self._total_sample_count = torch.zeros(1)
        self._curr_sample_count = torch.zeros(1)

    def __repr__(self):
        # implement what information about meter params should be
        # printed by print(meter). This is helpful for debugging
        return repr({"name": self.name, "value": self.value})

    def set_classy_state(self, state):
        """
        Set the state of meter. This is the state loaded from a checkpoint when the␣
↪model
        is resumed
        """
        # implement how to set the state of the meter
        ....

    def get_classy_state(self):
        """
        Returns the states of meter that will be checkpointed. This should include
        the variables that are global, updated and affect meter value.
        """
        return {
            "name": self.name,
            ...
        }

    def update(self, model_output, target):
        """
        Update the meter every time meter is calculated
        """
        # implement how to update the meter values
        ...

    def validate(self, model_output, target):
        """
        Validate that the input to meter is valid
```

(continues on next page)

(continued from previous page)

```
        """
        # implement how to enforce the validity of the meter inputs
        ....
```

- **Step2**: Register the meter and meter params with VISSL Configuration. Add the params that the meter takes in VISSL defaults.yaml as follows:

```
METERS:
  my_new_meter:
    param1: value1
    param2: value2
    ...
```

- **Step3**: Meter is ready to use. Simply set the configuration param `METERS.name=my_new_meter`

CHAPTER

FORTYSIX

ADD NEW MODELS

VISSL allows adding new models (head and trunks easily) and combine different trunks and heads to train a new model. Follow the steps below on how to add new heads or trunks.

46.1 Adding New Heads

To add a new model head, follow the steps:

- **Step1**: Add the new head `my_new_head` under `vissl/models/heads/my_new_head.py` following the template:

```
import torch
import torch.nn as nn
from vissl.models.heads import register_model_head

@register_model_head("my_new_head")
class MyNewHead(nn.Module):
    """
    Add documentation on what this head does and also link any papers where the head␣
↪is used
    """

    def __init__(self, model_config: AttrDict, param1: val, ....):
        """
        Args:
            add documentation on what are the parameters to the head
        """
        super().__init__()
        # implement what the init of head should do. Example, it can construct the␣
↪layers in the head
        # like FC etc., initialize the parameters or anything else
        ....

    # the input to the model should be a torch Tensor or list of torch tensors.
    def forward(self, batch: torch.Tensor or List[torch.Tensor]):
        """
        add documentation on what the head input structure should be, shapes expected
        and what the output should be
        """
        # implement the forward pass of the head
```

- **Step2**: The new head is ready to use. Test it by setting the new head in the configuration file.

```
MODEL:
  HEAD:
    PARAMS: [
        ...
        ["my_new_head", {"param1": val, ...}]
        ...
    ]
```

46.2 Adding New Trunks

To add a new trunk (a new architecture like vision transformers, etc.), follow the steps:

- **Step1**: Add your new trunk `my_new_trunk` under `vissl/data/trunks/my_new_trunk.py` following the template:

```
import torch
import torch.nn as nn
from vissl.models.trunks import register_model_trunk

@register_model_trunk("my_new_trunk")
class MyNewTrunk(nn.Module):
    """
    documentation on what the trunk does and links to technical reports
    using this trunk (if applicable)
    """

    def __init__(self, model_config: AttrDict, model_name: str):
        super(MyNewTrunk, self).__init__()
        self.model_config = model_config

        # get the params trunk takes from the config
        trunk_config = self.model_config.TRUNK.TRUNK_PARAMS.MyNewTrunk

        # implement the model trunk and construct all the layers that the trunk uses
        model_layer1 = ??
        model_layer2 = ??
        ...
        ...

        # give a name to the layers of your trunk so that these features
        # can be used for other purposes: like feature extraction etc.
        # the name is fully upto user descretion. User may chose to
        # only name one layer which is the last layer of the model.
        self._feature_blocks = nn.ModuleDict(
            [
                ("my_layer1_name", model_layer1),
                ("my_layer1_name", model_layer2),
                ...
            ]
        )

    def forward(
        self, x: torch.Tensor, out_feat_keys: List[str] = None
    ) -> List[torch.Tensor]:
        # implement the forward pass of the model. See the forward pass of resnext.py
```

(continues on next page)

(continued from previous page)

```
        # for reference.
        # The output would be a list. The list can have one tensor (the trunk output)
        # or mutliple tensors (corresponding to several features of the trunk)
        ...
        ...

        return output
```

- **Step2**: Inform VISSL about the parameters of the trunk. Register the params with VISSL Configuration by adding the params in VISSL defaults.yaml as follows:

```
MODEL:
  TRUNK:
    MyNewTrunk:
      param1: value1
      param2: value2
      ...
```

- **Step3**: The trunk is ready to use. Set the trunk name and params in your config file `MODEL.TRUNK.NAME=my_new_trunk`

CHAPTER

FORTYSEVEN

USING CUSTOM DATASETS

VISSL allows adding custom datasets easily. Using a new custom dataset has 2 requirements:

- **Requirement1**: The dataset name must be registered with `VisslDatasetCatalog`.
- **Requirement2**: Users should ensure that the data source is supported by VISSL. By default, VISSL supports reading data from disk. If user data is loaded from a different data source, please add the new data source following the documentation.

Follow the steps below to register and use the new dataset:

- **Step1**: Register the dataset with VISSL. Given user dataset with dataset name `my_new_dataset_name` and path to the dataset train and test splits, users can register the dataset following:

```
from vissl.data.dataset_catalog import VisslDatasetCatalog

VisslDatasetCatalog.register_data(name="my_new_dataset_name", data_dict={"train": ...␣
↪, "test": ...})
```

Note: VISSL also supports registering the dataset via a custom `json` file, or registering `dict` with bunch of datasets.

- **Step2 (Optional)**: If the dataset requires a new data source other than disk or supported disk formats (`disk_folder` or `disk_filelist`), please add the new data source to VISSL. Follow our documentation on Adding new dataset.
- **Step3**: Test your dataset

```
DATA:
  TRAIN:
    DATA_SOURCES: [my_data_source]
    DATASET_NAMES: [my_new_dataset_name]
```

CHAPTER

FORTYEIGHT

ADD NEW DATA SOURCE

VISSL supports data loading from `disk` as the default data source. If users dataset lives in their custom data storage solution `my_data_source` instead of `disk`, then users can extend VISSL to work with their data storage. Follow the steps below:

- **Step1**: Implement your custom data source under `vissl/data/my_data_source.py` following the template:

```
from vissl.data.data_helper import get_mean_image
from torch.utils.data import Dataset

class MyNewSourceDataset(Dataset):
    """
    add documentation on how this dataset works

    Args:
        add docstrings for the parameters
    """

    def __init__(self, cfg, data_source, path, split, dataset_name):
        super(MyNewSourceDataset, self).__init__()
        assert data_source in [
            "disk_filelist",
            "disk_folder",
            "my_data_source"
        ], "data_source must be either disk_filelist or disk_folder or my_data_source"
        self.cfg = cfg
        self.split = split
        self.dataset_name = dataset_name
        self.data_source = data_source
        self._path = path
        # implement anything that data source init should do
        ....
        ....
        self._num_samples = ?? # set the length of the dataset


    def num_samples(self):
        """
        Size of the dataset
        """
        return self._num_samples

    def __len__(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Size of the dataset
        """
        return self.num_samples()

    def __getitem__(self, idx: int):
        """
        implement how to load the data corresponding to idx element in the dataset
        from your data source
        """
        ....
        ....

        # is_success should be True or False indicating whether loading data was␣
↪successful or failed
        # loaded data should be Image.Image if image data
        return loaded_data, is_success
```

- **Step2**: Register the new data source with VISSL. Extend the `DATASET_SOURCE_MAP` dict in `vissl/data/__init__.py`.

```
DATASET_SOURCE_MAP = {
    "disk_filelist": DiskImageDataset,
    "disk_folder": DiskImageDataset,
    "synthetic": SyntheticImageDataset,
    "my_data_source": MyNewSourceDataset,
}
```

- **Step3**: Register the name of the datasets you plan to load using the new data source. There are 2 ways to do this:
  - See our documentation on "Using dataset_catalog.json" to update the `configs/dataset_catalog.json` file.
  - Insert a python call following:

```
# insert the following call in your python code
from vissl.data.dataset_catalog import VisslDatasetCatalog

VisslDatasetCatalog.register_data(name="my_dataset_name", data_dict={"train":␣
↪... , "test": ...})
```

- **Step4**: Test using your dataset

```
DATA:
  TRAIN:
    DATA_SOURCES: [my_data_source]
    DATASET_NAMES: [my_dataset_name]
```

CHAPTER

FORTYNINE

ADD NEW DATALOADER

VISSL currently supports PyTorch `torch.utils.data.DataLoader`. If users would like to add a custom dataloader of their own, we recommend the following steps.

- **Step1**: Create your custom dataloader class `MyNewDataLoader` in `vissl/data/my_loader.py`. The Dataloader should implement all the variables and member that PyTorch Dataloader uses.
- **Step2**: Import your new `MyNewDataLoader` in `vissl/data/__init__.py` and extend the function `get_loader(...)` to use your `MyNewDataLoader`. To control this from configuration file, we recommend users to add some config file options in `vissl/defaults.yaml` file under `DATA.TRAIN.dataloader_name`.

We welcome PRs following our Contributing guidelines.

CHAPTER

FIFTY

ADD NEW DATA TRANSFORMS

Adding new transforms and using them is quite easy in VISSL. Follow the steps below:

- **Step1**: Create your transform under `vissl/data/ssl_transforms/my_new_transform.py`. The transform should follow the template:

```
@register_transform("MyNewTransform")
class MyNewTransform(ClassyTransform):
    """
    add documentation for what your transform does
    """

    def __init__(self, param1, param2, ...):
        """
        Args:
            param1: add doctring
            param2: add doctring
            ...
        """
        self.param1 = param1
        self.param2 = param2
        # implement anything that the transform init should do
        ...

    # the input image should either be Image.Image PIL instance or torch.Tensor
    def __call__(self, image: {Image.Image or torch.Tensor}):
        # implement the transformation logic code.
        return img

    @classmethod
    def from_config(cls, config: Dict[str, Any]) -> "MyNewTransform":
        """
        Instantiates MyNewTransform from configuration.

        Args:
            config (Dict): arguments for for the transform

        Returns:
            MyNewTransform instance.
        """
        param1 = config.param1
        param2 = config.param2
        ....
        return cls(param1=param1, param2=param2, ...)
```

- **Step2**: Use your transform in the config file by editing the `DATA.TRAIN.TRANSFORMS` value:

```
DATA:
  TRANSFORMS:
    ...
    ...
    - name: MyNewTransform
      param1: value1
      param2: value2
      ....
    ....
```

CHAPTER

FIFTYONE

ADD NEW DATA COLLATORS

VISSL allows implementing new data collators easily. Follow the steps below:

- **Step1**: Create the new data collator `my_new_collator.py` under `vissl/data/collators/my_new_collator.py` following the template.

```
import torch
from vissl.data.collators import register_collator

@register_collator("my_new_collator")
def my_new_collator(batch, param1 (Optional), ...):
    """
    add documentation on what new collator does

    Input:
        add documentation on what input type should the collator expect. i.e
        what should the `batch' look like.

    Output:
        add documentation on what the collator returns i.e. what does the
        collated data `output_batch' look like.
    """
    # implement the collator
    ...
    ...

    output_batch = {
        "data": ... ,
        "label": ... ,
        "data_valid": ... ,
        "data_idx": ... ,
    }
    return output_batch
```

- **Step2**: Use your new collator via the configuration files

```
DATA:
  TRAIN:
    COLLATE_FUNCTION: my_new_collator
    COLLATE_FUNCTION_PARAMS: {...}  # optional, specify params if collator requires␣
↪any
```

CHAPTER

FIFTYTWO

ACTIVATION CHECKPOINTING TO REDUCE MODEL MEMORY

Authors: m1n@fb.com, lefaudeux@fb.com

Activation checkpointing is a very powerful technique to reduce the memory requirement of a model. This is especially useful when training very large models with billions of parameters.

52.1 How it works?

Activation checkpointing trades compute for memory. It discards intermediate activations during the forward pass, and recomputes them during the backward pass. In our experiments, using activation checkpointing, we observe negligible compute overhead in memory-bound settings while getting big memory savings.

In summary, This technique offers 2 benefits:

- saves gpu memory that can be used to fit large models
- allows increasing training batch size for a given model

We recommend users to read the documentation available here for further details on activation checkpointing.

52.2 How to use activation checkpointing in VISSL?

VISSL integrates activation checkpointing implementation directly from PyTorch available here. Using activation checkpointing in VISSL is extremely easy and doable with simple settings in the configuration file. The settings required are as below:

```
MODEL:
  ACTIVATION_CHECKPOINTING:
    # whether to use activation checkpointing or not
    USE_ACTIVATION_CHECKPOINTING: True
    # how many times the model should be checkpointed. User should tune this parameter
    # and find the number that offers best memory saving and compute tradeoff.
    NUM_ACTIVATION_CHECKPOINTING_SPLITS: 8
DISTRIBUTED:
  # if True, does the gradient reduction in DDP manually. This is useful during the
  # activation checkpointing and sometimes saving the memory from the pytorch gradient
  # buckets.
  MANUAL_GRADIENT_REDUCTION: True
```

CHAPTER

FIFTYTHREE

LARC FOR LARGE BATCH SIZE TRAINING

53.1 What is LARC

LARC (Large Batch Training of Convolutional Networks) is a technique proposed by **Yang You, Igor Gitman, Boris Ginsburg** in https://arxiv.org/abs/1708.03888 for improving the convergence of large batch size trainings. LARC uses the ratio between gradient and parameter magnitudes is used to calculate an adaptive local learning rate for each individual parameter.

See the LARC paper for calculation of learning rate. In practice, it modifies the gradients of parameters as a proxy for modifying the learning rate of the parameters.

53.2 How to enable LARC

VISSL supports the LARC implementation from NVIDIA's Apex LARC. To use LARC, users need to set config option `OPTIMIZER.use_larc=True`. VISSL exposes LARC parameters that users can tune. Full list of LARC parameters exposed by VISSL:

```
OPTIMIZER:
  name: "sgd"
  use_larc: False  # supported for SGD only for now
  larc_config:
    clip: False
    eps: 1e-08
    trust_coefficient: 0.001
```

Note: LARC is currently supported for SGD optimizer only.

53.2.1 Using Apex

In order to use Apex, VISSL provides `anaconda` and `pip` packages of Apex (compiled with Optimzed C++ extensions/CUDA kernels). The Apex packages are provided for all versions of `CUDA (9.2, 10.0, 10.1, 10.2, 11.0), PyTorch >= 1.4 and Python >=3.6 and <=3.9`.

Follow VISSL's instructions to install apex in pip and instructions to install apex in conda.

CHAPTER

FIFTYFOUR

HANDLING INVALID IMAGES IN DATALOADER

54.1 How VISSL solves it

Self-supervised approaches like SimCLR, SwAV etc that perform some form of contrastive learning contrast the features or cluster of one image with the other. During the dataloading time, or in the training dataset itself, it's possible that there are invalid images. By default, in VISSL, when the dataloader encounters an invalid image, a gray image is returned instead. Using gray images for the purpose of contrastive learning can lead to inferior model accuracy especially if there are a lot of invalid images.

To solve this issue, VISSL provides a custom *base* dataset class called `QueueDataset` that maintains 2 queues in CPU memory. One queue is used to enqueue valid seen images from previous minibatches and the other queue is used to dequeue. The code:*QueueDataset* is implemented such that the same minibatch will never have the duplicate images. If we can't dequeue a valid image, we return None from the dequeue. In short, code:*QueueDataset* enables using the previously used valid images from the training in the current minibatch in place of invalid images.

54.2 Enabling QueueDataset

VISSL makes it convenient for users to use the code:*QueueDataset* with simple configuration settings. To use the code:*QueueDataset*, users need to set `DATA.TRAIN.ENABLE_QUEUE_DATASET=true` and `DATA.TEST.ENABLE_QUEUE_DATASET=true`.

54.2.1 Tuning the queue size of QueueDataset

VISSL exposes the queue settings to configuration file that users can tune. The configuration settings are:

```
DATA:
  TRAIN:
    ENABLE_QUEUE_DATASET: True
  TEST:
    ENABLE_QUEUE_DATASET: True
```

Note: If users encounter CPU out-of-memory issue, they might want to reduce the queue size

CHAPTER

FIFTYFIVE

RESUME TRAINING FROM ITERATION: STATEFUL DATA SAMPLER

55.1 Issue with PyTorch DataSampler for large data training

PyTorch default torch.utils.data.distributed.DistributedSampler is the default sampler used for many trainings. However, it becomes limiting to use this sampler in case of large batch size trainings for 2 reasons:

- Using PyTorch `DataSampler`, each trainer shuffles the full data (assuming shuffling is used) and then each trainer gets a view of this shuffled data. If the dataset is large (100 millions, 1 billion or more), generating very large permutationon each trainer can lead to large CPU memory consumption per machine. Hence, it becomes difficult to use the PyTorch default `DataSampler` when user wants to train on large data and for several epochs (for example: 10 epochs of 100M images).
- When using PyTorch `DataSampler` and the training is resumed, the sampler will serve the full dataset. However, in case of large data trainings (like 1 billion images or more), one mostly trains for 1 epoch only. In such cases, when the training resumes from the middle of the epoch, the sampler will serve the full 1 billion images which is not what we want.

To solve both the above issues, VISSL provides a custom samplier `StatefulDistributedSampler` which inherits from the PyTorch *DistributedSampler* and fixes the above issues in following manner:

- Sampler creates the view of the data per trainer and then shuffles only the data that trainer is supposed to view. This keeps the CPU memory requirement expected.
- Sampler adds a member `start_iter` which tracks what iteration number of the given epoch model is at. When the training is used, the `start_iter` will be properly set to the last iteration number and the sampler will serve only the remainder of data.

55.2 How to use VISSL custom DataSampler

Using VISSL provided custom samplier `StatefulDistributedSampler` is extremely easy and involves simply setting the correct configuration options as below:

```
DATA:
  TRAIN:
    USE_STATEFUL_DISTRIBUTED_SAMPLER: True
  TEST:
    USE_STATEFUL_DISTRIBUTED_SAMPLER: True
```

Note: Users can use `StatefulDistributedSampler` for only training dataset and use PyTorch default `DataSampler` if desired i.e. it is not mandatory to use the same sampler type for all data splits.

CHAPTER

FIFTYSIX

MIXED PRECISION TRAINING (FP16)

Many self-supervised approaches leverage mixed precision training by default for better training speed and reducing the model memory requirement. For this, we use NVIDIA Apex Library with AMP.

Users can tune the AMP level to the levels supported by NVIDIA. See this for details on Apex amp levels.

To use Mixed precision training, one needs to set the following parameters in configuration file:

```
MODEL:
  AMP_PARAMS:
    USE_AMP: True
    # Use O1 as it is robust and stable than O3. If you want to use O3, we recommend
    # the following setting:
    # {"opt_level": "O3", "keep_batchnorm_fp32": True, "master_weights": True, "loss_
↪scale": "dynamic"}
    AMP_ARGS: {"opt_level": "O1"}
```

56.1 Using Apex

In order to use Apex, VISSL provides `anaconda` and `pip` packages of Apex (compiled with Optimzed C++ extensions/CUDA kernels). The Apex packages are provided for all versions of `CUDA (9.2, 10.0, 10.1, 10.2, 11.0), PyTorch >= 1.4 and Python >=3.6 and <=3.9`.

Follow VISSL's instructions to install apex in pip and instructions to install apex in conda.

CHAPTER

FIFTYSEVEN

TRAIN ON MULTIPLE-GPUS

VISSL supports training any model on 1-gpu or more. Typically, a single machine can have 2, 4 or 8-gpus. If users want to train on >1 gpus within the single machine, it's very easy. Typically for single machine training, this involves correctly setting the number of gpus to use via `DISTRIBUTED.NUM_PROC_PER_NODE`.

The config will look like:

```
DISTRIBUTED:
  BACKEND: nccl            # set to "gloo" if desired
  NUM_NODES: 1             # no change needed
  NUM_PROC_PER_NODE: 2     # user sets this to number of gpus to use
  INIT_METHOD: tcp         # set to "file" if desired
  RUN_ID: auto             # Set to file_path if using file method. No change needed␣
↪for tcp and a free port on machine is automatically detected.
```

The list of all the options exposed by VISSL:

```
DISTRIBUTED:
  # backend for communication across gpus. Use nccl by default. For cpu training, set
  # "gloo" as the backend.
  BACKEND: "nccl"
  # whether to output the NCCL info during training. This allows to debug how
  # nccl communication is configured.
  NCCL_DEBUG: False
  # tuning parameter to speed up all reduce by specifying number of nccl threads to␣
↪use.
  # by default, we use whatever the default is set by nccl or user system.
  NCCL_SOCKET_NTHREADS: ""
  # whether model buffers are BN buffers are broadcast in every forward pass
  BROADCAST_BUFFERS: True
  # number of machines to use in training. Each machine can have many gpus. NODES␣
↪count
  # number of unique hosts.
  NUM_NODES: 1
  # set this to the number of gpus per machine. This ensrures that each gpu of the
  # node has a process attached to it.
  NUM_PROC_PER_NODE: 8
  # this could be: tcp | env | file or any other pytorch supported methods
  INIT_METHOD: "tcp"
  # every training run should have a unique id. Following are the options:
  #   1. If using INIT_METHOD=env, RUN_ID="" is fine.
  #   2. If using INIT_METHOD=tcp,
  #      - if you use > 1 machine, set port yourself. RUN_ID="localhost:{port}".
  #      - If using 1 machine, set RUN_ID=auto and a free port will be automatically␣
↪selected
```

(continues on next page)

(continued from previous page)

```
  #   3. IF using INIT_METHOD=file, RUN_ID={file_path}
  RUN_ID: "auto"
```

CHAPTER

FIFTYEIGHT

TRAIN ON MULTIPLE MACHINES

VISSL allows scaling a training beyond 1-machine in order to speed up training. VISSL makes it extremely easy to scale up training. Typically for single machine training, this involves correctly setting the following options:

- Number of gpus to use
- Number of nodes
- INIT_METHOD for PyTorch distributed training which determines how gpus will communicate for all reduce operations.

Putting togethe the above, if user wants to train on 2 machines where each machine has 8-gpus, the config will look like:

```
DISTRIBUTED:
  BACKEND: nccl
  NUM_NODES: 2                # user sets this to number of machines to use
  NUM_PROC_PER_NODE: 8        # user sets this to number of gpus to use per machine
  INIT_METHOD: tcp            # recommended if feasible otherwise
  RUN_ID: localhost:{port}    # select the free port
```

The list of all the options exposed by VISSL:

```
DISTRIBUTED:
  # backend for communication across gpus. Use nccl by default. For cpu training, set
  # "gloo" as the backend.
  BACKEND: "nccl"
  # whether to output the NCCL info during training. This allows to debug how
  # nccl communication is configured.
  NCCL_DEBUG: False
  # tuning parameter to speed up all reduce by specifying number of nccl threads to␣
↪use.
  # by default, we use whatever the default is set by nccl or user system.
  NCCL_SOCKET_NTHREADS: ""
  # whether model buffers are BN buffers are broadcast in every forward pass
  BROADCAST_BUFFERS: True
  # number of machines to use in training. Each machine can have many gpus. NODES␣
↪count
  # number of unique hosts.
  NUM_NODES: 1
  # set this to the number of gpus per machine. This ensrures that each gpu of the
  # node has a process attached to it.
  NUM_PROC_PER_NODE: 8
  # this could be: tcp | env | file or any other pytorch supported methods
  INIT_METHOD: "tcp"
  # every training run should have a unique id. Following are the options:
```

(continues on next page)

(continued from previous page)

```
  #   1. If using INIT_METHOD=env, RUN_ID="" is fine.
  #   2. If using INIT_METHOD=tcp,
  #      - if you use > 1 machine, set port yourself. RUN_ID="localhost:{port}".
  #      - If using 1 machine, set RUN_ID=auto and a free port will be automatically␣
↪selected
  #   3. IF using INIT_METHOD=file, RUN_ID={file_path}
  RUN_ID: "auto"
```

CHAPTER

FIFTYNINE

USING SLURM

VISSL supports SLURM by default for training models. VISSL code automatically detects if the training environment is SLURM based on SLURM environment variables like `SLURM_NODEID`, `SLURMD_NODENAME`, `SLURM_STEP_NODELIST`.

VISSL also provides a helper bash script dev/launch_slurm.sh that allows launching a given training on SLURM. Users can modify this script to meet their needs.

The bash script takes the following inputs:

```
# number of machines to distribute training on
NODES=${NODES-1}
# number of gpus per machine to use for training
NUM_GPU=${NUM_GPU-8}
# gpus type: P100 | V100 | V100_32G etc. User should set this based on their machine
GPU_TYPE=${GPU_TYPE-V100}
# name of the training. for example: simclr_2node_resnet50_in1k. This is helpful to␣
↪clearly recognize the training
EXPT_NAME=${EXPT_NAME}
# how much CPU memory to use
MEM=${MEM-250g}
# number of CPUs used for each trainer (i.e. each gpu)
CPU=${CPU-8}
# directory where all the training artifacts like checkpoints etc will be written
OUTPUT_DIR=${OUTPUT_DIR}
# partition of the cluster on which training should run. User should determine this␣
↪parameter for their cluster
PARTITION=${PARTITION-learnfair}
# any helpful comment that slurm dashboard can display
COMMENT=${COMMENT-vissl_training}
GITHUB_REPO=${GITHUB_REPO-vissl}
# what branch of VISSL should be used. specify your custom branch
BRANCH=${BRANCH-master}
# automatically determined and used for distributed training.
# each training run must have a unique id and vissl defaults to date
RUN_ID=$(date +'%Y%m%d')
# number of dataloader workers to use per gpu
NUM_DATA_WORKERS=${NUM_DATA_WORKERS-8}
# multi-processing method to use in PyTorch. Options: forkserver | fork | spawn
MULTI_PROCESSING_METHOD=${MULTI_PROCESSING_METHOD-forkserver}
# specify the training configuration to run. For example: to train swav for 100epochs
# config=pretrain/swav/swav_8node_resnet config.OPTIMIZER.num_epochs=100
CFG=( "$@" )
```

To run the script for training SwAV on 8 machines where each machine has 8-gpus and for 100epochs, the script can be run as:

```
cd $HOME/vissl && NODES=8 \
  NUM_GPU=8 \
  GPU_TYPE=V100 \
  MEM=200g \
  CPU=8 \
  EXPT_NAME=swav_100ep_rn50_in1k \
  OUTPUT_DIR=/tmp/swav/ \
  PARTITION=learnfair \
  BRANCH=master \
  NUM_DATA_WORKERS=4 \
  MULTI_PROCESSING_METHOD=forkserver \
  ./dev/launch_slurm.sh \
  config=pretrain/swav/swav_8node_resnet config.OPTIMIZER.num_epochs=100
```

CHAPTER

SIXTY

ZERO: OPTIMIZER STATE AND GRADIENT SHARDING

Author: lefaudeux@fb.com

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models is a technique developed by **Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, Yuxiong He** in this paper. When training models with billions of parameters, GPU memory becomes a bottleneck. ZeRO can offer 4x to 8x memory reductions in memory thus allowing to fit larger models in memory.

60.1 How ZeRO works?

Memory requirement of a model can be broken down roughly into:

1. activations memory
2. model parameters
3. parameters momentum buffers (optimizer state)
4. parameters gradients

ZeRO *shards* the optimizer state and the parameter gradients onto different devices and reduces the memory needed per device.

60.2 How to use ZeRO in VISSL?

VISSL uses FAIRScale library which implements ZeRO in PyTorch. Using VISSL in ZeRO involves no code changes and can simply be done by setting some configuration options in the yaml files.

In order to use ZeRO, user needs to set `OPTIMIZER.name=zero` and nest the desired optimizer (for example SGD) settings in `OPTIMIZER.base_optimizer`.

An example for using ZeRO with LARC and SGD optimization:

```
OPTIMIZER:
  name: zero
  base_optimizer:
    name: sgd
    use_larc: False
    larc_config:
      clip: False
      trust_coefficient: 0.001
      eps: 0.00000001
    weight_decay: 0.000001
```

(continues on next page)

(continued from previous page)

```
    momentum: 0.9
    nesterov: False
```

Note: ZeRO works seamlessly with LARC and mixed precision training. Using ZeRO with activation checkpointing is not yet enabled primarily due to manual gradient reduction need for activation checkpointing.

CHAPTER

SIXTYONE

API DOCUMENTATION

61.1 vissl.data package

class vissl.data.**GenericSSLDataset**(*cfg*, *split*, *dataset_source_map*)

Bases: torch.utils.data.dataset.Dataset

Base Self Supervised Learning Dataset Class.

The GenericSSLDataset class is defined to support reading data from multiple data sources. For example: data = [dataset1, dataset2] and the minibatches generated will have the corresponding data from each dataset.

For this reason, we also support labels from multiple sources. For example targets = [dataset1 targets, dataset2 targets].

In order to support multiple data sources, the dataset configuration always has list inputs.

- DATA_SOURCES, LABEL_SOURCES, DATASET_NAMES, DATA_PATHS, LABEL_PATHS

For several data sources, we also support specifying on what dataset the transforms should be applied. By default, apply the transforms on data from all datasets.

Parameters

- **cfg** (AttrDict) – configuration defined by user
- **split** (*str*) – the dataset split for which we are constructing the Dataset object
- **dataset_source_map** (*Dict[str, Callable]*) – The dictionary that maps what data sources are supported and what object to use to read data from those sources. For example: DATASET_SOURCE_MAP = {

  ”disk_filelist”: DiskImageDataset, “disk_folder”: DiskImageDataset, “synthetic”: SyntheticImageDataset,

  }

load_single_label_file(*path*)

Load the single data file. We only support user specifying the numpy label files if user is specifying a data_filelist source of labels.

To save memory, if the mmap_mode is set to True for loading, we try to load the images in mmap_mode. If it fails, we simply load the labels without mmap

__getitem__(*idx*)

Get the input sample for the minibatch for a specified data index. For each data object (if we are loading several datasets in a minibatch), we get the sample: consisting of {

- image data,
- label (if applicable) otherwise idx

- data_valid: 0 or 1 indicating if the data is valid image
- data_idx : index of the data in the dataset for book-keeping and debugging

}

Once the sample data is available, we apply the data transform on the sample.

The final transformed sample is returned to be added into the minibatch.

__len__()
Size of the dataset. Assumption made there is only one data source

get_image_paths()
Get the image paths for all the data sources.

Returns

image_paths (List[List[str]]) –

list containing image paths list for each data source.

get_available_splits(*dataset_config*)
Get the available splits in the dataset confir. Not specific to this split for which the SSLDataset is being constructed.

NOTE: this is deprecated method.

num_samples(*source_idx=0*)
Size of the dataset. Assumption made there is only one data source

get_batchsize_per_replica()
Get the batch size per trainer

get_global_batchsize()
The global batch size across all the trainers

vissl.data.**get_data_files**(*split*, *dataset_config*)

Get the path to the dataset (images and labels).

1. If the user has explicitly specified the data_sources, we simply use those and don't do lookup in the datasets registered with VISSL from the dataset catalog.
2. If the user hasn't specified the path, look for the dataset in the datasets catalog registered with VISSL. For a given list of datasets and a given partition (train/test), we first verify that we have the dataset and the correct source as specified by the user. Then for each dataset in the list, we get the data path (make sure it exists, sources match). For the label file, the file is optional.

Once we have the dataset original paths, we replace the path with the local paths if the data was copied to local disk.

vissl.data.**register_datasets**(*json_catalog_path*)
If the json dataset_catalog file is found, we register the datasets specified in the catalog with VISSL. If the catalog also specified VOC or coco datasets, we resister them

Parameters **json_catalog_path** (*str*) – the path to the json dataset catalog

class vissl.data.**VisslDatasetCatalog**
Bases: object

A catalog that stores information about the datasets and how to obtain them. It contains a mapping from strings (which are names that identify a dataset, e.g. "imagenet1k") to a *dict* which contains:

1) mapping of various data splits (train, test, val) to the data source (path on the disk whether a folder path or a filelist)

2) source of the data (disk_filelist | disk_folder)

The purpose of having this catalog is to make it easy to choose different datasets, by just using the strings in the config.

static register_json(*json_catalog_path*)

Parameters **filepath** – a .json filepath that contains the data to be registered

static register_dict(*dict_catalog*)

Parameters **dict** – a dict with a bunch of datasets to be registered

static register_data(*name*, *data_dict*)

Parameters

- **name** (*str*) – the name that identifies a dataset, e.g. "imagenet1k_folder".
- **func** (*callable*) – a callable which takes no arguments and returns a list of dicts. It must return the same results if called multiple times.

static get(*name*)
Get the registered dict and return it.

Parameters **name** (*str*) – the name that identifies a dataset, e.g. "imagenet1k".

Returns *dict* – dataset information (paths, source)

static list() → List[str]
List all registered datasets.

Returns list[str]

static clear()
Remove all registered dataset.

static remove(*name*)
Remove the dataset registered by `name`.

static has_data(*name*)
Check whether the data with `name` exists.

61.1.1 vissl.data.collators module

vissl.data.collators.**register_collator**(*name*)
Registers Self-Supervision data collators.

This decorator allows VISSL to add custom data collators, even if the collator itself is not part of VISSL. To use it, apply this decorator to a collator function, like this:

```
@register_collator('my_collator_name')
def my_collator_name():
    ...
```

To get a collator from a configuration file, see *get_collator()*.

vissl.data.collators.**get_collator**(*collator_name*, *collate_params*)
Given the collator name and the collator params, return the collator if registered with VISSL. Also supports pytorch default collators.

61.1.2 vissl.data.collators.mixup_collator module

vissl.data.collators.mixup_collator.**multicrop_mixup_collator**(*batch*)

This collator is used to mix-up 2 images at a time. 2*N input images becomes N images This collator can handle multi-crop input. For each crop, it mixes-up the corresponding crop of the next image.

Input:

batch: Example

batch = [{"data" : [img1_0, . . . , img1_k], ..}, {"data" : [img2_0, . . . , img2_k], . . . }, . . . {"data" : [img2N_0, . . . , img2N_k], . . . },

]

Returns: Example output:

output = [

{

"data": [torch.tensor([img1_2_0, . . . , img1_2_k]), torch.tensor([img3_4_0, . . . , img3_4_k])

. . .

]

},

]

61.1.3 vissl.data.collators.moco_collator module

vissl.data.collators.moco_collator.**moco_collator**(*batch: List[Dict[str, Any]]*) → Dict[str, List[torch.Tensor]]

This collator is specific to MoCo approach http://arxiv.org/abs/1911.05722

The collators collates the batch for the following input (assuming k-copies of image):

Input:

batch: Example

batch = [{"data" : [img1_0, . . . , img1_k], ..}, {"data" : [img2_0, . . . , img2_k], . . . }, . . .

]

Returns: Example output:

output = [

{ "data": torch.tensor([img1_0, . . . , img1_k], [img2_0, . . . , img2_k]) ..

},

]

Dimensions become [num_positives x Batch x C x H x W]

61.1.4 vissl.data.collators.multicrop_collator module

vissl.data.collators.multicrop_collator.**multicrop_collator**(*batch*)
This collator is used in SwAV approach.

The collators collates the batch for the following input (assuming k-copies of image):

Input:

batch: Example

batch = [{"data" : [img1_0, . . . , img1_k], ..}, {"data" : [img2_0, . . . , img2_k], . . . }, . . .

]

Returns: Example output:

output = [

{ "data": torch.tensor([img1_0, . . . , imgN_0], [img1_k, . . . , imgN_k]) ..

},

]

61.1.5 vissl.data.collators.patch_and_image_collator module

vissl.data.collators.patch_and_image_collator.**patch_and_image_collator**(*batch*)
This collator is used in PIRL approach.

batch contains two keys "data" and "label".

- data is a list of N+1 elements. 1st element is the "image" and remainder N are patches.
- label is an integer (image index in the dataset)

We collate this to image: batch_size tensor containing images patches: N * batch_size tensor containing patches

61.1.6 vissl.data.collators.siamese_collator module

vissl.data.collators.siamese_collator.**siamese_collator**(*batch*)
This collator is used in Jigsaw approach.

Input:

batch: Example

batch = [{"data": [img1,], "label": [lbl1,]}, #img1 {"data": [img2,], "label": [lbl2,]}, #img2 . . {"data": [imgN,], "label": [lblN,]}, #imgN

]

where: img{x} is a tensor of size: num_towers x C x H x W lbl{x} is an integer

Returns: Example output:

output = [

{ "data": torch.tensor([img1_0, . . . , imgN_0]) ..

},

] where the output is of dimension: (N * num_towers) x C x H x W

61.1.7 vissl.data.collators.simclr_collator module

vissl.data.collators.simclr_collator.**simclr_collator**(*batch*)

This collator is used in SimCLR approach.

The collators collates the batch for the following input (each image has k-copies): input: [[img1_0, ..., img1_k], [img2_0, ..., img2_k], ..., [imgN_0, ..., imgN_k]] output: [img1_0, img2_0, ..., img1_1, img2_1,...]

Input:

batch: Example

batch = [{"data": [img1_0, ..., img1_k], "label": [lbl1,]}, #img1 {"data": [img2_0, ..., img2_k], "label": [lbl2,]}, #img2 . . {"data": [imgN_0, ..., imgN_k], "label": [lblN,]}, #imgN

]

where: img{x} is a tensor of size: C x H x W lbl{x} is an integer

Returns: Example output:

output = [

{ "data": torch.tensor([img1_0, img2_0, ..., img1_1, img2_1,...]) ..

},

]

61.1.8 vissl.data.collators.targets_one_hot_default_collator module

vissl.data.collators.targets_one_hot_default_collator.**convert_to_one_hot**(*pos_lbl*, *neg_lbl*, *num_classes: int*) → torch.Tensor

This function converts target class indices to one-hot vectors, given the number of classes.

-> 1 for positive labels, -> 0 for negative and -> -1 for ignore labels.

vissl.data.collators.targets_one_hot_default_collator.**targets_one_hot_default_collator**(*batch*, *num_*, *int*)

The collators collates the batch for the following input:

Input: input : [[img0, ..., imgk]] label: [

[[1, 3, 6], [4, 9]] [[1, 5], [6, 8, 10, 11]] .....

]

Output: output: [img0, img0, ...,] label: [[0, 1, 0, 1, ..., -1, 0, 0, 1], [0, 1, 0, 0, 0, 1, 0], ...]

61.1.9 vissl.data.ssl_transforms module

class vissl.data.ssl_transforms.**SSLTransformsWrapper**(*indices*, ***args*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

VISSL wraps around transforms so that they work with the multimodal input. VISSL supports batches that come from several datasets and sources. Hence the input batch (images, labels) always is a list.

To apply the user defined transforms, VISSL takes "indices" as input which defines on what dataset/source data in the sample should the transform be applied to. For example:

Assuming input sample is { "data": [dataset1_imgX, dataset2_imgY], "label": [dataset1_lblX, dataset2_lblY]

} and the transform is:

TRANSFORMS:

- name: RandomGrayscale p: 0.2 indices: 0

then the transform is applied only on dataset1_imgX. If however, the indices are either not specified or set to 0, 1 then the transform is applied on both dataset1_imgX and dataset2_imgY

Since this structure of data is introduced by vissl, the SSLTransformsWrapper takes care of dealing with the multi-modality input by wrapping the original transforms (pytorch transforms or custom transforms defined by user) and calling each transform on each index.

VISSL also supports _TRANSFORMS_WITH_LABELS transforms that modify the label or are used to generate the labels used in self-supervised learning tasks like Jigsaw. When the transforms in _TRANSFORMS_WITH_LABELS are called, the new label is also returned besides the transformed image.

VISSL also supports the _TRANSFORMS_WITH_COPIES which are transforms that basically generate several copies of image. Common example of self-supervised training methods that do this is SimCLR, SwAV, MoCo etc When a transform from _TRANSFORMS_WITH_COPIES is used, the SSLTransformsWrapper will flatten the transform output. For example for the input [img1], if we apply ImgReplicatePil to replicate the image 2 times:

SSLTransformsWrapper(ImgReplicatePil(num_times=2), [img1]

) will output [img1_1, img1_2] instead of nested list [[img1_1, img1_2]].

The benefit of this is that the next set of transforms specified by user can now operate on img1_1 and img1_2 as the input becomes multi-modal nature.

VISSL also supports _TRANSFORMS_WITH_GROUPING which essentially means that a single transform should be applied on the full multi-modal input together instead of separately. This is common transform used in BYOL/ For example:

SSLTransformsWrapper(

ImgPilMultiCropRandomApply(RandomApply, prob=[0.0, 0.2]

), [img1_1, img1_2]

) this will apply RandomApply on img1_1 with prob=0.0 and on img1_2 with prob=0.2

__init__(*indices*, ***args*)

Parameters

- **indices** (*List[int]) (Optional*) – the indices list on which transform should be applied for the input which is always a list Example: minibatch of size=2 looks like [[img1], [img2]]). If indices is not specified, transform is applied to all the multi-modal input.

- **args** (*dict*) – the arguments that the transform takes

__call__(*sample*)
Apply each transform on the specified indices of each entry in the input sample.

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.SSLTransformsWrapper*

vissl.data.ssl_transforms.**get_transform**(*input_transforms_list*)
Given the list of user specified transforms, return the torchvision.transforms.Compose() version of the transforms. Each transform in the composition is SSLTransformsWrapper which wraps the original transforms to handle multi-modal nature of input.

61.1.10 vissl.data.ssl_transforms.img_patches_tensor module

class vissl.data.ssl_transforms.img_patches_tensor.**ImgPatchesFromTensor**(*num_patches=9*, *patch_jitter=21*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Create image patches from a torch Tensor or numpy array. This transform was proposed in Jigsaw - https://arxiv.org/abs/1603.09246

Parameters

- **num_patches** (*int*) – how many image patches to create
- **patch_jitter** (*int*) – space to leave between patches

__call__(*image*)
Input image which is a torch.Tensor object of shape 3 x H x W

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_patches_tensor.ImgPatchesFromTens*
Instantiates ImgPatchesFromTensor from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPatchesFromTensor instance.

61.1.11 vissl.data.ssl_transforms.img_pil_color_distortion module

class vissl.data.ssl_transforms.img_pil_color_distortion.**ImgPilColorDistortion**(*strength*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Apply Random color distortions to the input image. There are multiple different ways of applying these distortions. This implementation follows SimCLR - https://arxiv.org/abs/2002.05709 It randomly distorts the hue, saturation, brightness of an image and can randomly convert the image to grayscale.

__init__(*strength*)

Parameters **strength** (*float*) – A number used to quantify the strength of the color distortion.

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_color_distortion.ImgPilColorDis*
Instantiates ImgPilColorDistortion from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilColorDistortion instance.

61.1.12 vissl.data.ssl_transforms.img_pil_gaussian_blur module

class vissl.data.ssl_transforms.img_pil_gaussian_blur.**ImgPilGaussianBlur**(*p*, *radius_min*, *radius_max*)

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Apply Gaussian Blur to the PIL image. Take the radius and probability of application as the parameter.

This transform was used in SimCLR - https://arxiv.org/abs/2002.05709

__init__(*p*, *radius_min*, *radius_max*)

Parameters

- **p** (*float*) – probability of applying gaussian blur to the image
- **radius_min** (*float*) – blur kernel minimum radius used by ImageFilter.GaussianBlur
- **radius_max** (*float*) – blur kernel maximum radius used by ImageFilter.GaussianBlur

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_gaussian_blur.ImgPilGaussianB*

Instantiates ImgPilGaussianBlur from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilGaussianBlur instance.

61.1.13 vissl.data.ssl_transforms.img_pil_multicrop_random_apply module

class vissl.data.ssl_transforms.img_pil_multicrop_random_apply.**ImgPilMultiCropRandomApply**(*t*

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Apply a list of transforms on multi-crop input. The transforms are Randomly applied to each crop using the specified probability. This is used in BYOL https://arxiv.org/pdf/2006.07733.pdf

Multi-crops are several crops of a given image. This is most commonly used in contrastive learning. For example SimCLR, SwAV approaches use multi-crop input.

__init__(*transforms: List[Dict[str, Any]]*, *prob: float*)

Parameters

- **transforms** (*List (tranforms)*) – List of transforms that should be applied to each crop.
- **prob** (*List (float)*) – Probability of RandomApply for the transforms composition on each crop. example: for 2 crop in BYOL, for solarization:

  prob = [0.0, 0.2]

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_multicrop_random_apply.ImgPil*

Instantiates ImgPilMultiCropRandomApply from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilMultiCropRandomApply instance.

61.1.14 vissl.data.ssl_transforms.img_pil_random_color_jitter module

class vissl.data.ssl_transforms.img_pil_random_color_jitter.**ImgPilRandomColorJitter**(*strength*, *prob*)

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Apply Random color jitter to the input image. It randomly distorts the hue, saturation, brightness of an image.

__init__(*strength*, *prob*)

Parameters

- **strength** (*float*) – A number used to quantify the strength of the color distortion.
- **p** (*float*) – probability of random application

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_random_color_jitter.ImgPilRand*

Instantiates ImgPilRandomColorJitter from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilRandomColorJitter instance.

61.1.15 vissl.data.ssl_transforms.img_pil_random_photometric module

class vissl.data.ssl_transforms.img_pil_random_photometric.**ImgPilRandomPhotometric**(*p*)

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Randomly apply some photometric transforms to an image. This was used in PIRL - https://arxiv.org/abs/1912.01991

The photometric transforms applied includes: AutoContrast, RandomPosterize, RandomSharpness, RandomSolarize

__init__(*p*)

Parameters **p** (*float*) – Probability of applying the transforms

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_random_photometric.ImgPilRan*

Instantiates ImgPilRandomPhotometric from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilRandomPhotometric instance.

61.1.16 vissl.data.ssl_transforms.img_pil_random_solarize module

class vissl.data.ssl_transforms.img_pil_random_solarize.**ImgPilRandomSolarize**(*prob: float*)

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Randomly apply solarization transform to an image. This was used in BYOL - https://arxiv.org/abs/2006.07733

__init__(*prob: float*)

Parameters **p** (*float*) – Probability of applying the transform

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_random_solarize.ImgPilRandom*

Instantiates ImgPilRandomSolarize from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilRandomSolarize instance.

61.1.17 vissl.data.ssl_transforms.img_pil_to_lab_tensor module

class vissl.data.ssl_transforms.img_pil_to_lab_tensor.**ImgPil2LabTensor**(*indices*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Convert a PIL image to LAB tensor of shape C x H x W This transform was proposed in Colorization - https://arxiv.org/abs/1603.08511

The input image is PIL Image. We first convert it to tensor HWC which has channel order RGB. We then convert the RGB to BGR and use OpenCV to convert the image to LAB. The LAB image is 8-bit image in range > L [0, 255], A [0, 255], B [0, 255]. We rescale it to: L [0, 100], A [-128, 127], B [-128, 127]

The output is image torch tensor.

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_to_lab_tensor.ImgPil2LabTensor*
Instantiates ImgPil2LabTensor from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPil2LabTensor instance.

61.1.18 vissl.data.ssl_transforms.img_pil_to_multicrop module

class vissl.data.ssl_transforms.img_pil_to_multicrop.**ImgPilToMultiCrop**(*total_num_crops*, *num_crops*, *size_crops*, *crop_scales*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Convert a PIL image to Multi-resolution Crops. The input is a PIL image and output is the list of image crops.

This transform was proposed in SwAV - https://arxiv.org/abs/2006.09882

__init__(*total_num_crops*, *num_crops*, *size_crops*, *crop_scales*)
Returns total_num_crops square crops of an image. Each crop is a random crop extracted according to the parameters specified in size_crops and crop_scales. For ease of use, one can specify *num_crops* which removes the need to repeat parameters.

Parameters

- **total_num_crops** (*int*) – Total number of crops to extract
- **num_crops** (*List or Tuple of ints*) – Specifies the number of `type' of crops.
- **size_crops** (*List or Tuple of ints*) – Specifies the height (height = width) of each patch
- **crop_scales** (*List or Tuple containing [float, float]*) – Scale of the crop

Example usage: - (total_num_crops=2, num_crops=[1, 1],

size_crops=[224, 96], crop_scales=[(0.14, 1.), (0.05, 0.14)]) Extracts 2 crops total of size 224x224 and 96x96

- **(total_num_crops=2, num_crops=[1, 2],** size_crops=[224, 96], crop_scales=[(0.14, 1.), (0.05, 0.14)]) Extracts 3 crops total: 1 of size 224x224 and 2 of size 96x96

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_to_multicrop.ImgPilToMultiCrop*
Instantiates ImgPilToMultiCrop from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilToMultiCrop instance.

61.1.19 vissl.data.ssl_transforms.img_pil_to_patches_and_image module

class vissl.data.ssl_transforms.img_pil_to_patches_and_image.**ImgPilToPatchesAndImage**(*crop_sc*
1.0,
crop_siz
crop_sc
1.0,
crop_siz
per-
mute_pa
num_pa

Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Convert an input PIL image to Patches and Image This transform was proposed in PIRL - https://arxiv.org/abs/1912.01991.

Input: PIL Image

Returns

list containing N+1 elements

- zeroth element: a RandomResizedCrop of the image
- remainder: N patches extracted uniformly from a RandomResizedCrop

__init__(*crop_scale_image=0.08, 1.0, crop_size_image=224, crop_scale_patches=0.6, 1.0, crop_size_patches=255, permute_patches=True, num_patches=9*)

Parameters

- **crop_scale_image** (*tuple of floats*) – scale for RandomResizedCrop of image
- **crop_size_image** (*int*) – size for RandomResizedCrop of image
- **crop_scale_patches** (*tuple of floats*) – scale for RandomResizedCrop of patches
- **crop_size_patches** (*int*) – size for RandomResizedCrop of patches
- **permute_patches** (*bool*) – permute the patches in any order
- **num_patches** (*int*) – number of patches to create. should be a square integer.

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_to_patches_and_image.ImgPilTo*
Instantiates ImgPilToPatchesAndImage from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilToPatchesAndImage instance.

61.1.20 vissl.data.ssl_transforms.img_pil_to_raw_tensor module

class vissl.data.ssl_transforms.img_pil_to_raw_tensor.**ImgPilToRawTensor**
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Convert a PIL image to Raw Tensor if we don't want to apply the default division by 255 by torchvision.transforms.ToTensor()

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_pil_to_raw_tensor.ImgPilToRawTens*
Instantiates ImgPilToRawTensor from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgPilToRawTensor instance.

61.1.21 vissl.data.ssl_transforms.img_pil_to_tensor module

class vissl.data.ssl_transforms.img_pil_to_tensor.**ImgToTensor**
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

The Transform that overrides the PyTorch transform to provide better transformation speed.

credits: mannatsingh@fb.com

61.1.22 vissl.data.ssl_transforms.img_replicate_pil module

class vissl.data.ssl_transforms.img_replicate_pil.**ImgReplicatePil**(*num_times: int = 2*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Adds the same image multiple times to the batch K times so that the batch. Size is now N*K. Use the simclr_collator to convert into batches.

This transform is useful when generating multiple copies of the same image, for example, when training contrastive methods.

__init__(*num_times: int = 2*)

Parameters **num_times** (*int*) – how many times should the image be replicated.

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_replicate_pil.ImgReplicatePil*
Instantiates ImgReplicatePil from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgReplicatePil instance.

61.1.23 vissl.data.ssl_transforms.img_rotate_pil module

class vissl.data.ssl_transforms.img_rotate_pil.**ImgRotatePil**(*num_angles=4, num_rotations_per_img=1*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

Apply rotation to a PIL Image. Samples rotation angle from a set of predefined rotation angles.

Predefined rotation angles are sampled at equal intervals in the [0, 360) angle space where the number of angles is specified by *num_angles*.

This transform was used in RotNet - https://arxiv.org/abs/1803.07728

__init__ (*num_angles=4*, *num_rotations_per_img=1*)

Parameters

- **num_angles** (*int*) – Number of angles in the [0, 360) space
- **num_rotations_per_img** (*int*) – Number of rotations to apply to each image.

classmethod from_config (*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.img_rotate_pil.ImgRotatePil*
Instantiates ImgRotatePil from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ImgRotatePil instance.

61.1.24 vissl.data.ssl_transforms.pil_photometric_transforms_lib module

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**TransformObject**
Bases: object

Helper object to that prints information about the transformation and other transforms can inherit from this.

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**RandomValueApplier**(*min_v*, *max_v*, *root_transfo*, *vtype='float'*, *closed_inter*)
Bases: *vissl.data.ssl_transforms.pil_photometric_transforms_lib.TransformObject*

__init__ (*min_v*, *max_v*, *root_transform*, *vtype='float'*, *closed_interval=False*)
Applies a transform by sampling a random value between [min_v, max_v]

Parameters

- **min_v** (*float or int*) – minimum value
- **max_v** (*float or int*) – maximum value
- **root_transform** (*transform object*) – transform that will be applied. must accept a value as input.
- **vtype** (*string*) – value type - either "float" or "int"
- **closed_interval** (*bool*) – sample from [min_v, max_v] (when True) or [min_v, max_v) when False

sample_value ()
Randomly sample the value from min_v and max_v depending on float or int type and also whether to use open or closed interval for sampleing

vissl.data.ssl_transforms.pil_photometric_transforms_lib.**Sharpness** (*img*, *v*)
Applies PIL.ImageEnhance.Sharpness to the image

vissl.data.ssl_transforms.pil_photometric_transforms_lib.**Solarize** (*img*, *v*)
Applies PIL.ImageOps.solarize to the image

vissl.data.ssl_transforms.pil_photometric_transforms_lib.**Posterize** (*img*, *v*)
Applies PIL.ImageOps.posterize to the image

vissl.data.ssl_transforms.pil_photometric_transforms_lib.**AutoContrast** (*img*, *_*)
Applies PIL.ImageOps.autocontrast to the image

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**RandomSharpnessTransform**(*min_v*, *max_v*, *root_transform*, *vtype*)

Bases: *vissl.data.ssl_transforms.pil_photometric_transforms_lib.RandomValueApplier*

Randomly apply the Sharpness transformation with the random value selected from an interval.

__init__(*min_v=0.1*, *max_v=1.9*, *root_transform=<function Sharpness>*, *vtype='float'*)

Parameters

- **min_v** (*float*) – minimum value
- **max_v** (*float*) – maximum value
- **root_transform** (*transform object*) – transform that will be applied. must accept a value as input.
- **vtype** (*string*) – value type - "float"

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**RandomPosterizeTransform**(*min_v*, *max_v*, *root_transform*, *vtype*)

Bases: *vissl.data.ssl_transforms.pil_photometric_transforms_lib.RandomValueApplier*

__init__(*min_v=4*, *max_v=8*, *root_transform=<function Posterize>*, *vtype='int'*)

Parameters

- **min_v** (*int*) – minimum value
- **max_v** (*int*) – maximum value
- **root_transform** (*transform object*) – transform that will be applied. must accept a value as input.
- **vtype** (*string*) – value type - "int"

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**RandomSolarizeTransform**(*min_v*, *max_v*, *root_transform*, *vtype*)

Bases: *vissl.data.ssl_transforms.pil_photometric_transforms_lib.RandomValueApplier*

__init__(*min_v=0*, *max_v=256*, *root_transform=<function Solarize>*, *vtype='int'*)

Parameters

- **min_v** (*int*) – minimum value
- **max_v** (*int*) – maximum value

- **root_transform** (*transform object*) – transform that will be applied. must accept a value as input.
- **vtype** (*string*) – value type - "int"

class vissl.data.ssl_transforms.pil_photometric_transforms_lib.**AutoContrastTransform**
Bases: *vissl.data.ssl_transforms.pil_photometric_transforms_lib.TransformObject*

Wraps the AutoContrast method

61.1.25 vissl.data.ssl_transforms.shuffle_img_patches module

class vissl.data.ssl_transforms.shuffle_img_patches.**ShuffleImgPatches**(*perm_file: str*)
Bases: classy_vision.dataset.transforms.classy_transform.ClassyTransform

This transform is used to shuffle the list of tensors (usually image patches of shape C x H x W) according to a randomly selected permutation from a pre-defined set of permutations.

This is a common operation used in Jigsaw approach https://arxiv.org/abs/1603.09246

__init__(*perm_file: str*)

Parameters **perm_file** (*string*) – path to the file containing pre-defined permutations.

__call__(*input_patches*)
The interface *__call__* is used to transform the input data. It should contain the actual implementation of data transform.

Parameters **input_patches** (*List[torch.tensor]*) – list of torch tensors

classmethod from_config(*config: Dict[str, Any]*) → *vissl.data.ssl_transforms.shuffle_img_patches.ShuffleImgPatches*
Instantiates ShuffleImgPatches from configuration.

Parameters **config** (*Dict*) – arguments for for the transform

Returns ShuffleImgPatches instance.

61.1.26 vissl.data.data_helper module

vissl.data.data_helper.**get_mean_image**(*crop_size*)
Helper function that returns a gray PIL image of the size specified by user.

Parameters **crop_size** (*int*) – used to generate (crop_size x crop_size x 3) image.

Returns *img* – PIL Image

class vissl.data.data_helper.**StatefulDistributedSampler**(*dataset*, *batch_size=None*)
Bases: torch.utils.data.distributed.DistributedSampler

More fine-grained state DataSampler that uses training iteration and epoch both for shuffling data. PyTorch DistributedSampler only uses epoch for the shuffling and starts sampling data from the start. In case of training on very large data, we train for one epoch only and when we resume training, we want to resume the data sampler from the training iteration.

__init__(*dataset*, *batch_size=None*)
Initializes the instance of StatefulDistributedSampler. Random seed is set for the epoch set and data is shuffled. For starting the sampling, use the start_iter (set to 0 or set by checkpointing resuming) to sample data from the remaining images.

Parameters

- **dataset** (*Dataset*) – Pytorch dataset that sampler will shuffle
- **batch_size** (*int*) – batch size we want the sampler to sample

set_start_iter(*start_iter*)
Set the iteration number from which the sampling should start. This is used to find the marker in the data permutation order from where the sampler should start sampling.

class vissl.data.data_helper.**QueueDataset**(*queue_size*)
Bases: torch.utils.data.dataset.Dataset

This class helps dealing with the invalid images in the dataset by using two queue. One queue is used to enqueue seen and valid images from previous batches. The other queue is used to dequeue. The class is implemented such that the same batch will never have duplicate images. If we can't dequeue a valid image, we return None for that instance.

Parameters **queue_size** – size the the queue (ideally set it to batch_size). Both queues will be of the same size

on_sucess(*sample*)
If we encounter a successful image and the queue is not full, we store it in the queue. One consideration we make further is: if the image is very large, we don't add it to the queue as otherwise the CPU memory will grow a lot.

on_failure()
If there was a failure in getting the origin image, we look into the queue if there is any valid seen image available. If yes, we dequeue and use this image in place of the failed image.

61.1.27 vissl.data.dataloader_sync_gpu_wrapper module

class vissl.data.dataloader_sync_gpu_wrapper.**DataloaderSyncGPUWrapper**(*dataloader: Iterable*)
Bases: classy_vision.dataset.dataloader_wrapper.DataloaderWrapper

Dataloader which wraps another dataloader, and moves the data to GPU in async manner so as to overlap the cost of copying data from cpu to gpu with the previous model iteration.

61.1.28 vissl.data.ssl_dataset module

class vissl.data.ssl_dataset.**GenericSSLDataset**(*cfg*, *split*, *dataset_source_map*)
Bases: torch.utils.data.dataset.Dataset

Base Self Supervised Learning Dataset Class.

The GenericSSLDataset class is defined to support reading data from multiple data sources. For example: data = [dataset1, dataset2] and the minibatches generated will have the corresponding data from each dataset.

For this reason, we also support labels from multiple sources. For example targets = [dataset1 targets, dataset2 targets].

In order to support multiple data sources, the dataset configuration always has list inputs.

- DATA_SOURCES, LABEL_SOURCES, DATASET_NAMES, DATA_PATHS, LABEL_PATHS

For several data sources, we also support specifying on what dataset the transforms should be applied. By default, apply the transforms on data from all datasets.

Parameters

- **cfg** (AttrDict) – configuration defined by user
- **split** (*str*) – the dataset split for which we are constructing the Dataset object
- **dataset_source_map** (*Dict[str, Callable]*) – The dictionary that maps what data sources are supported and what object to use to read data from those sources. For example: DATASET_SOURCE_MAP = {

  "disk_filelist": DiskImageDataset, "disk_folder": DiskImageDataset, "synthetic": SyntheticImageDataset,

  }

load_single_label_file(*path*)

Load the single data file. We only support user specifying the numpy label files if user is specifying a data_filelist source of labels.

To save memory, if the mmap_mode is set to True for loading, we try to load the images in mmap_mode. If it fails, we simply load the labels without mmap

__getitem__(*idx*)

Get the input sample for the minibatch for a specified data index. For each data object (if we are loading several datasets in a minibatch), we get the sample: consisting of {

- image data,
- label (if applicable) otherwise idx
- data_valid: 0 or 1 indicating if the data is valid image
- data_idx : index of the data in the dataset for book-keeping and debugging

}

Once the sample data is available, we apply the data transform on the sample.

The final transformed sample is returned to be added into the minibatch.

__len__()

Size of the dataset. Assumption made there is only one data source

get_image_paths()

Get the image paths for all the data sources.

Returns

image_paths (List[List[str]]) –

list containing image paths list for each data source.

get_available_splits(*dataset_config*)

Get the available splits in the dataset confir. Not specific to this split for which the SSLDataset is being constructed.

NOTE: this is deprecated method.

num_samples(*source_idx=0*)

Size of the dataset. Assumption made there is only one data source

get_batchsize_per_replica()

Get the batch size per trainer

get_global_batchsize()

The global batch size across all the trainers

61.1.29 vissl.data.disk_dataset module

class vissl.data.disk_dataset.**DiskImageDataset**(*cfg*, *data_source*, *path*, *split*, *dataset_name*)

Bases: *vissl.data.data_helper.QueueDataset*

Base Dataset class for loading images from Disk. Can load a predefined list of images or all images inside a folder.

Inherits from QueueDataset class in VISSL to provide better handling of the invalid images by replacing them with the valid and seen images.

Parameters

- **cfg** (AttrDict) – configuration defined by user
- **data_source** (*string*) – data source either of "disk_filelist" or "disk_folder"
- **path** (*string*) – can be either of the following 1. A .npy file containing a list of filepaths.

  In this case *data_source = "disk_filelist"*

  2. A folder such that folder/split contains images. In this case *data_source = "disk_folder"*

- **split** (*string*) – specify split for the dataset. Usually train/val/test. Used to read images if reading from a folder *path* and retrieve settings for that split from the config path.
- **dataset_name** (*string*) – name of dataset. For information only.

NOTE: This dataset class only returns images (not labels or other metdata). To load labels you must specify them in *LABEL_SOURCES* (See *ssl_dataset.py*). LABEL_SOURCES follows a similar convention as the dataset and can either be a filelist or a torchvision ImageFolder compatible folder - 1. Store labels in a numpy file 2. Store images in a nested directory structure so that torchvision ImageFolder

dataset can infer the labels.

num_samples()
Size of the dataset

get_image_paths()
Get paths of all images in the datasets. See load_data()

__len__()
Size of the dataset

__getitem__(*idx*)

- We do delayed loading of data to reduce the memory size due to pickling of dataset across dataloader workers.
- Loads the data if not already loaded.
- Sets and initializes the queue if not already initialized
- Depending on the data source (folder or filelist), get the image. If using the QueueDataset and image is valid, save the image in queue if not full. Otherwise return a valid seen image from the queue if queue is not empty.

61.1.30 vissl.data.synthetic_dataset module

class vissl.data.synthetic_dataset.**SyntheticImageDataset**(*cfg*, *path*, *split*, *dataset_name*, *data_source='synthetic'*)

Bases: torch.utils.data.dataset.Dataset

Synthetic dataset class. Mean image is returned always. This dataset is used/recommended to use for testing purposes only.

Parameters

- **path** (*string*) – can be "" [not used]
- **split** (*string*) – specify split for the dataset. Usually train/val/test. Used to read images if reading from a folder `path' and retrieve settings for that split from the config path [not used]
- **dataset_name** (*string*) – name of dataset. For information only. [not used]
- **data_source** (*string, Optional*) – data source ("synthetic") [not used]

num_samples()
Size of the dataset

__len__()
Size of the dataset

__getitem__(*idx*)
Simply return the mean dummy image of the specified size and mark it as a success.

61.1.31 vissl.data.dataset_catalog module

Data and labels file for various datasets.

class vissl.data.dataset_catalog.**VisslDatasetCatalog**
Bases: object

A catalog that stores information about the datasets and how to obtain them. It contains a mapping from strings (which are names that identify a dataset, e.g. "imagenet1k") to a *dict* which contains:

1) mapping of various data splits (train, test, val) to the data source (path on the disk whether a folder path or a filelist)
2) source of the data (disk_filelist | disk_folder)

The purpose of having this catalog is to make it easy to choose different datasets, by just using the strings in the config.

static register_json(*json_catalog_path*)

Parameters **filepath** – a .json filepath that contains the data to be registered

static register_dict(*dict_catalog*)

Parameters **dict** – a dict with a bunch of datasets to be registered

static register_data(*name*, *data_dict*)

Parameters

- **name** (*str*) – the name that identifies a dataset, e.g. "imagenet1k_folder".

- **func** (*callable*) – a callable which takes no arguments and returns a list of dicts. It must return the same results if called multiple times.

static get(*name*)
Get the registered dict and return it.

Parameters **name** (*str*) – the name that identifies a dataset, e.g. "imagenet1k".

Returns *dict* – dataset information (paths, source)

static list() → List[str]
List all registered datasets.

Returns list[str]

static clear()
Remove all registered dataset.

static remove(*name*)
Remove the dataset registered by `name`.

static has_data(*name*)
Check whether the data with `name` exists.

vissl.data.dataset_catalog.**get_local_path**(*input_file*, *dest_dir*)
If user specified copying data to a local directory, get the local path where the data files were copied.

- If input_file is just a file, we return the dest_dir/filename
- If the intput_file is a directory, then we check if the environemt is SLURM and use slurm_dir or otherwise dest_dir to look up copy_complete file is available. If available, we return the directory.
- If both above fail, we return the input_file as is.

vissl.data.dataset_catalog.**get_local_output_filepaths**(*input_files*, *dest_dir*)
If we have copied the files to local disk as specified in the config, we return those local paths. Otherwise return the original paths.

vissl.data.dataset_catalog.**check_data_exists**(*data_files*)
Check that the input data files exist. If the data_files is a list, we iteratively check for each file in the list.

vissl.data.dataset_catalog.**register_pascal_voc**()
Register PASCAL VOC 2007 and 2012 datasets to the data catalog. We first look up for these datasets paths in the dataset catalog, if the paths exist, we register, otherwise we remove the voc_data from the catalog registry.

vissl.data.dataset_catalog.**register_coco**()
Register COCO 2004 datasets to the data catalog. We first look up for these datasets paths in the dataset catalog, if the paths exist, we register, otherwise we remove the coco2014_folder from the catalog registry.

vissl.data.dataset_catalog.**register_datasets**(*json_catalog_path*)
If the json dataset_catalog file is found, we register the datasets specified in the catalog with VISSL. If the catalog also specified VOC or coco datasets, we resister them

Parameters **json_catalog_path** (*str*) – the path to the json dataset catalog

vissl.data.dataset_catalog.**get_data_files**(*split*, *dataset_config*)

Get the path to the dataset (images and labels).

1. If the user has explicitly specified the data_sources, we simply use those and don't do lookup in the datasets registered with VISSL from the dataset catalog.
2. If the user hasn't specified the path, look for the dataset in the datasets catalog registered with VISSL. For a given list of datasets and a given partition (train/test), we first verify that we have the dataset

and the correct source as specified by the user. Then for each dataset in the list, we get the data path (make sure it exists, sources match). For the label file, the file is optional.

Once we have the dataset original paths, we replace the path with the local paths if the data was copied to local disk.

61.2 vissl.engines package

61.2.1 vissl.engines.train module

vissl.engines.train.**train_main**(*cfg: vissl.utils.hydra_config.AttrDict, dist_run_id: str, checkpoint_path: str, checkpoint_folder: str, local_rank: int = 0, node_id: int = 0, hook_generator: Callable[[Any], List[classy_vision.hooks.classy_hook.ClassyHook]] = <function default_hook_generator>*)

Sets up and executes training workflow per machine.

Parameters

- **cfg** (AttrDict) – user specified input config that has optimizer, loss, meters etc settings relevant to the training
- **dist_run_id** (*str*) – For multi-gpu training with PyTorch, we have to specify how the gpus are going to rendezvous. This requires specifying the communication method: file, tcp and the unique rendezvous run_id that is specific to 1 run. We recommend:

  1) for 1node: use init_method=tcp and run_id=auto

  2) for multi-node, use init_method=tcp and specify run_id={master_node}:{port}
- **checkpoint_path** (*str*) – if the training is being resumed from a checkpoint, path to the checkpoint. The tools/run_distributed_engines.py automatically looks for the checkpoint in the checkpoint directory.
- **checkpoint_folder** (*str*) – what directory to use for checkpointing. The tools/run_distributed_engines.py creates the directory based on user input in the yaml config file.
- **local_rank** (*int*) – id of the current device on the machine. If using gpus, local_rank = gpu number on the current machine
- **node_id** (*int*) – id of the current machine. starts from 0. valid for multi-gpu
- **hook_generator** (*Callable*) – The utility function that prepares all the hoooks that will be used in training based on user selection. Some basic hooks are used by default.

61.2.2 vissl.engines.extract_features module

vissl.engines.extract_features.**extract_main**(*cfg: vissl.utils.hydra_config.AttrDict, dist_run_id: str, local_rank: int = 0, node_id: int = 0*)

Sets up and executes feature extraction workflow per machine.

Parameters

- **cfg** (AttrDict) – user specified input config that has optimizer, loss, meters etc settings relevant to the training

- **dist_run_id** (*str*) – For multi-gpu training with PyTorch, we have to specify how the gpus are going to rendezvous. This requires specifying the communication method: file, tcp and the unique rendezvous run_id that is specific to 1 run. We recommend:

  1) for 1node: use init_method=tcp and run_id=auto

  2) for multi-node, use init_method=tcp and specify run_id={master_node}:{port}

- **local_rank** (*int*) – id of the current device on the machine. If using gpus, local_rank = gpu number on the current machine
- **node_id** (*int*) – id of the current machine. starts from 0. valid for multi-gpu

61.3 vissl.meters package

61.3.1 vissl.meters.accuracy_list_meter

class vissl.meters.accuracy_list_meter.**AccuracyListMeter**(*num_meters: int*, *topk_values: List[int]*, *meter_names: List[str]*)

Bases: classy_vision.meters.classy_meter.ClassyMeter

Meter to calculate top-k accuracy for single label image classification task.

Supports Single target and multiple output. A list of accuracy meters is constructed and each output has a meter associated.

Parameters

- **num_meters** – number of meters and hence we have same number of outputs
- **topk_values** – list of int *k* values. Example: [1, 5]
- **meter_names** – list of str indicating the name of meter. Usually corresponds to the output layer name.

classmethod from_config(*meters_config:* vissl.utils.hydra_config.AttrDict)
Get the AccuracyListMeter instance from the user defined config

property name
Name of the meter

property value
Value of the meter globally synced. For each output, all the top-k values are returned. If there are several meters attached to the same layer name, a list of top-k values will be returned for that layer name meter.

sync_state()
Globally syncing the state of each meter across all the trainers.

get_classy_state()
Returns the states of each meter

set_classy_state(*state*)
Set the state of each meter

update(*model_output: Union[torch.Tensor, List[torch.Tensor]]*, *target: torch.Tensor*)
Updates the value of the meter for the given model output list and targets.

Parameters

- **model_output** – list of tensors of shape (B, C) where each value is either logit or class probability.

- **target** – tensor of shape (B).

NOTE: For binary classification, C=2.

reset()
Reset all the meters

validate(*model_output_shape*, *target_shape*)
Not implemented

61.3.2 vissl.meters.mean_ap_meter

class vissl.meters.mean_ap_meter.**MeanAPMeter**(*meters_config:* vissl.utils.hydra_config.AttrDict)

Bases: classy_vision.meters.classy_meter.ClassyMeter

Meter to calculate mean AP metric for multi-label image classification task.

Parameters **meters_config** (AttrDict) – config containing the meter settings

meters_config should specify the num_classes

classmethod from_config(*meters_config:* vissl.utils.hydra_config.AttrDict)
Get the AccuracyListMeter instance from the user defined config

property name
Name of the meter

property value
Value of the meter globally synced. mean AP and AP for each class is returned

gather_scores(*scores: torch.Tensor*)
Do a gather over all embeddings, so we can compute the loss. Final shape is like: (batch_size * num_gpus) x embedding_dim

gather_targets(*targets: torch.Tensor*)
Do a gather over all embeddings, so we can compute the loss. Final shape is like: (batch_size * num_gpus) x embedding_dim

sync_state()
Globally syncing the state of each meter across all the trainers. We gather scores, targets, total sampled

reset()
Reset the meter

set_classy_state(*state*)
Set the state of meter

get_classy_state()
Returns the states of meter

verify_target(*target*)
Verify that the target contains {-1, 0, 1} values only

update(*model_output*, *target*)
Update the scores and targets

validate(*model_output*, *target*)
Validate that the input to meter is valid

61.3.3 vissl.meters.mean_ap_list_meter

class vissl.meters.mean_ap_list_meter.**MeanAPListMeter**(*meters_config:* vissl.utils.hydra_config.AttrDict)

Bases: classy_vision.meters.classy_meter.ClassyMeter

Meter to calculate mean AP metric for multi-label image classification task on multiple output single target.

Supports Single target and multiple output. A list of mean AP meters is constructed and each output has a meter associated.

Parameters **meters_config** (AttrDict) – config containing the meter settings

meters_config should specify the num_meters and meter_names

classmethod from_config(*meters_config:* vissl.utils.hydra_config.AttrDict)
Get the AccuracyListMeter instance from the user defined config

property name
Name of the meter

property value
Value of the meter globally synced. For each output, mean AP and AP for each class is returned.

sync_state()
Globally syncing the state of each meter across all the trainers.

get_classy_state()
Returns the states of each meter

set_classy_state(*state*)
Set the state of each meter

update(*model_output: Union[torch.Tensor, List[torch.Tensor]]*, *target: torch.Tensor*)
Updates the value of the meter for the given model output list and targets.

Parameters

- **model_output** – list of tensors of shape (B, C) where each value is either logit or class probability.
- **target** – tensor of shape (B).

NOTE: For binary classification, C=2.

reset()
Reset all the meters

validate(*model_output_shape*, *target_shape*)
Not implemented

61.4 vissl.models package

class vissl.models.**BaseSSLMultiInputOutputModel**(**args*, ***kwargs*)

Bases: classy_vision.models.classy_model.ClassyModel

Class to implement a Self-Supervised model. The model is split into `trunk' that computes features and `head' that computes outputs (projections, classifications etc)

This class supports many use cases: 1. Model producing single output as in standard supervised ImageNet training 2. Model producing multiple outputs (Multi-task) 3. Model producing multiple outputs from different features (layers)

from the trunk (useful in linear evaluation of features from several model layers)

4. Model that accepts multiple inputs (e.g. image and patches as in PIRL appraoch).
5. Model where the trunk is frozen.
6. Model that supports multiple resolutions inputs as in SwAV

- **How to specify heads?** For information on heads see the *_get_heads()* function
- **What inputs do `heads' operate on?** One can specify the `input' to heads mapping in the list MULTI_INPUT_HEAD_MAPPING. See the _setup_multi_input_head_mapping() function for details.

multi_input_with_head_mapping_forward(*batch*)
Perform forward pass (trunk + heads) separately on each input and return the model output on all inputs as a list.

multi_res_input_forward(*batch*, *feature_names*)
Perform forward pass separately on each resolution input. The inputs corresponding to a single resolution are clubbed and single forward is run on the same resolution inputs. Hence we do several forward passes = number of different resolutions used. We then concatenate all the output features. Then run the head forward on the concatenated features.

single_input_forward(*batch*, *feature_names*, *heads*)
Simply run the trunk and heads forward on the input tensor. We run the trunk first and then the heads on the trunk output. If the model is trunk feature extraction only, then we simply return the output of the trunk.

heads_forward(*feats*, *heads*)
Run the forward of the head on the trunk output features. We have 2 cases:

1. #heads = #feats -> example training linear classifiers on various layers. We run one head on the corresponding feature.
2. #feats = 1 and #heads > 1 -> head consists of many layers to be run sequentially. #outputs = 1

forward(*batch*)
Main forward of the model. Depending on the model type the calls are patched to the suitable function.

freeze_head()
Freeze the model head by setting requires_grad=False for all the parameters

freeze_trunk()
Freeze the model trunk by setting requires_grad=False for all the parameters

freeze_head_and_trunk()
Freeze the full model including the heads and the trunk. In 99% cases, we do not use the pretext head as it is specific to the self-supervised pretext task. But in case of some models like NPID, SimCLR, SwAV, the head is essentially a low dimensional feature projection which we want to use. Hence, we provide utility to freeze the full model.

is_fully_frozen_model()
Look at all the parameters of the model (trunk + heads) and check if there is any trainable parameter. if not, the model is completely frozen.

get_features(*batch*)
Run the trunk forward on the input batch. This give us the features from the trunk at several layers of the model.

In case of feature extraction, we don't run the heads and only the trunk. The trunk will already have the feature extractor Pooling layers and flattened features attached. feature extractor heads are part of the trunk already.

get_classy_state(*deep_copy=False*)
Return the model state (trunk + heads) to checkpoint.

We call this on the state.base_model which is not wrapped with DDP. get the model state_dict to checkpoint

set_classy_state(*state*)
Initialize the model trunk and head from the state dictionary.

We call this on the state.base_model which is not wrapped with DDP. load the model from checkpoint.

property num_classes
Not implemented and not required

property input_shape
Not implemented and not required

property output_shape
Not implemented and not required

validate(*dataset_output_shape*)
Not implemented and not required

vissl.models.**convert_sync_bn**(*config*, *model*)
Convert the BatchNorm layers in the model to the SyncBatchNorm layers.

For SyncBatchNorm, we support two sources: Apex and PyTorch. The optimized SyncBN kernels provided by apex run faster.

Parameters

- **config** (AttrDict) – configuration file
- **model** – Pytorch model whose BatchNorm layers should be converted to SyncBN layers.

NOTE: Since SyncBatchNorm layer synchronize the BN stats across machines, using the syncBN layer can be slow. In order to speed up training while using syncBN, we recommend using process_groups which are very well supported for Apex. To set the process groups, set SYNC_BN_CONFIG.GROUP_SIZE following below: 1) if group_size=-1 -> use the VISSL default setting. We synchronize within a

machine and hence will set group_size=num_gpus per node. This gives the best speedup.

2) if group_size>0 -> will set group_size=value set by user.

3) if group_size=0 -> no groups are created and process_group=None. This means global sync is done.

vissl.models.**is_feature_extractor_model**(*model_config*)

If the model is a feature extractor model:

- evaluation model is on
- trunk is frozen
- number of features specified for features extratction > 0

vissl.models.**build_model**(*model_config*, *optimizer_config*)
Given the model config and the optimizer config, construct the model. The returned model is not copied to gpu yet (if using gpu) and neither wrapped with DDP yet. This is done later train_task.py .prepare()

61.4.1 vissl.models.model_helpers module

vissl.models.model_helpers.**transform_model_input_data_type**(*model_input*, *model_config*)
Default model input follow RGB format. Based the model input specified, change the type. Supported types: RGB, BGR, LAB

vissl.models.model_helpers.**is_feature_extractor_model**(*model_config*)

If the model is a feature extractor model:

- evaluation model is on
- trunk is frozen
- number of features specified for features extratction > 0

vissl.models.model_helpers.**get_trunk_output_feature_names**(*model_config*)
Get the feature names which we will use to associate the features witl. If Feature eval mode is set, we get feature names from config.FEATURE_EVAL_SETTINGS.LINEAR_EVAL_FEAT_POOL_OPS_MAP.

class vissl.models.model_helpers.**Wrap**(*function*)
Bases: torch.nn.modules.module.Module

Wrap a free function into a nn.Module. Can be useful to build a model block, and include activations or light tensor alterations

forward(*x*)

class vissl.models.model_helpers.**SyncBNTypes**(*value*)
Bases: str, enum.Enum

Supported SyncBN types

apex = 'apex'

pytorch = 'pytorch'

vissl.models.model_helpers.**convert_sync_bn**(*config*, *model*)
Convert the BatchNorm layers in the model to the SyncBatchNorm layers.

For SyncBatchNorm, we support two sources: Apex and PyTorch. The optimized SyncBN kernels provided by apex run faster.

Parameters

- **config** (AttrDict) – configuration file
- **model** – Pytorch model whose BatchNorm layers should be converted to SyncBN layers.

NOTE: Since SyncBatchNorm layer synchronize the BN stats across machines, using the syncBN layer can be slow. In order to speed up training while using syncBN, we recommend using process_groups which are very well supported for Apex. To set the process groups, set SYNC_BN_CONFIG.GROUP_SIZE following below: 1) if group_size=-1 -> use the VISSL default setting. We synchronize within a

machine and hence will set group_size=num_gpus per node. This gives the best speedup.

2) if group_size>0 -> will set group_size=value set by user.

3) if group_size=0 -> no groups are created and process_group=None. This means global sync is done.

class vissl.models.model_helpers.**Flatten**(*dim=- 1*)
Bases: torch.nn.modules.module.Module

Flatten module attached in the model. It basically flattens the input tensor.

forward(*feat*)
flatten the input feat

flops(*x*)
number of floating point operations performed. 0 for this module.

class vissl.models.model_helpers.**Identity**(*args=None*)
Bases: torch.nn.modules.module.Module

A helper module that outputs the input as is

forward(*x*)
Return the input as the output

class vissl.models.model_helpers.**LayerNorm2d**(*num_channels*, *eps=1e-05*, *affine=True*)
Bases: torch.nn.modules.normalization.GroupNorm

Use GroupNorm to construct LayerNorm as pytorch LayerNorm2d requires specifying input_shape explicitly which is inconvenient. Set num_groups=1 to convert GroupNorm to LayerNorm.

class vissl.models.model_helpers.**RESNET_NORM_LAYER**(*value*)
Bases: str, enum.Enum

Types of Norms supported in ResNe(X)t trainings. can be easily set and modified from the config file.

BatchNorm = 'BatchNorm'

LayerNorm = 'LayerNorm'

vissl.models.model_helpers.**parse_out_keys_arg**(*out_feat_keys: List[str]*, *all_feat_names: List[str]*) → Tuple[List[str], int]
Checks if all out_feature_keys are mapped to a layer in the model. Returns the last layer to forward pass through for efficiency. Allow duplicate features also to be evaluated. Adapted from (https://github.com/gidariss/FeatureLearningRotNet).

vissl.models.model_helpers.**get_trunk_forward_outputs_module_list**(*feat: torch.Tensor*, *out_feat_keys: List[str]*, *feature_blocks: torch.nn.modules.container.ModuleLis*, *all_feat_names: List[str] = None*) → List[torch.Tensor]

Parameters

- **feat** – model input.
- **out_feat_keys** – a list/tuple with the feature names of the features that the function should return. By default the last feature of the network is returned.
- **feature_blocks** – list of feature blocks in the model
- **feature_mapping** – name of the layers in the model

Returns *out_feats* – a list with the asked output features placed in the same order as in *out_feat_keys*.

vissl.models.model_helpers.**get_trunk_forward_outputs**(*feat: torch.Tensor*, *out_feat_keys: List[str]*, *feature_blocks: torch.nn.modules.container.ModuleDict*, *feature_mapping: Dict[str, str] = None*, *use_checkpointing: bool = True*, *checkpointing_splits: int = 2*) → List[torch.Tensor]

Parameters

- **feat** – model input.
- **out_feat_keys** – a list/tuple with the feature names of the features that the function should return. By default the last feature of the network is returned.
- **feature_blocks** – ModuleDict containing feature blocks in the model
- **feature_mapping** – an optional correspondence table in between the requested feature names and the model's.

Returns *out_feats* – a list with the asked output features placed in the same order as in *out_feat_keys*.

61.4.2 vissl.models.heads module

vissl.models.heads.**get_model_head**(*name: str*)

Given the model head name, construct the head if it's registered with VISSL.

class vissl.models.heads.**LinearEvalMLP**(*model_config: vissl.utils.hydra_config.AttrDict*, *in_channels: int*, *dims: List[int]*, *use_bn: bool = False*, *use_relu: bool = False*)

Bases: torch.nn.modules.module.Module

A standard Linear classification module that can be attached to several layers of the model to evaluate the representation quality of features.

The layers attached are: BatchNorm2d -> Linear (1 or more)

Accepts a 4D input tensor. If you want to use 2D input tensor instead, use the "mlp" head directly.

__init__(*model_config: vissl.utils.hydra_config.AttrDict*, *in_channels: int*, *dims: List[int]*, *use_bn: bool = False*, *use_relu: bool = False*)

Parameters

- **model_config** (*AttrDict*) – dictionary config.MODEL in the config file
- **in_channels** (*int*) – number of channels the input has. This information is used to attached the BatchNorm2D layer.
- **dims** (*int*) – dimensions of the linear layer. Example [8192, 1000] which means attaches *nn.Linear(8192, 1000, bias=True)*

forward(*batch: torch.Tensor*)

Parameters **batch** (*torch.Tensor*) – 4D torch tensor. This layer is meant to be attached at several parts of the model to evaluate feature representation quality. For 2D input tensor, the tensor is unsqueezed to NxDx1x1 and then eval_mlp is applied

Returns *out (torch.Tensor)* – 2D output torch tensor

class vissl.models.heads.**MLP**(*model_config:* vissl.utils.hydra_config.AttrDict, *dims:* *List[int]*, *use_bn:* *bool* = *False*, *use_relu:* *bool* = *False*, *use_dropout:* *bool* = *False*, *use_bias:* *bool* = *True*)

Bases: torch.nn.modules.module.Module

This module can be used to attach combination of {Linear, BatchNorm, Relu, Dropout} layers and they are fully configurable from the config file. The module also supports stacking multiple MLPs.

Examples

Linear Linear -> BN Linear -> ReLU Linear -> Dropout Linear -> BN -> ReLU -> Dropout Linear -> ReLU -> Dropout Linear -> ReLU -> Linear -> ReLU -> . . . Linear -> Linear -> . . . . . .

Accepts a 2D input tensor. Also accepts 4D input tensor of shape *N x C x 1 x 1*.

__init__(*model_config:* vissl.utils.hydra_config.AttrDict, *dims:* *List[int]*, *use_bn:* *bool* = *False*, *use_relu:* *bool* = *False*, *use_dropout:* *bool* = *False*, *use_bias:* *bool* = *True*)

Parameters

- **model_config** (*AttrDict*) – dictionary config.MODEL in the config file
- **use_bn** (*bool*) – whether to attach BatchNorm after Linear layer
- **use_relu** (*bool*) – whether to attach ReLU after (Linear (-> BN optional))
- **use_dropout** (*bool*) – whether to attach Dropout after (Linear (-> BN -> relu optional))
- **use_bias** (*bool*) – whether the Linear layer should have bias or not
- **dims** (*int*) – dimensions of the linear layer. Example [8192, 1000] which attaches *nn.Linear(8192, 1000, bias=True)*

scale_weights(*model_config*)

forward(*batch:* *torch.Tensor*)

Parameters **batch** (*torch.Tensor*) – 2D torch tensor or 4D tensor of shape *N x C x 1 x 1*

Returns *out (torch.Tensor)* – 2D output torch tensor

class vissl.models.heads.**SiameseConcatView**(*model_config:* vissl.utils.hydra_config.AttrDict, *num_towers:* *int*)

Bases: torch.nn.modules.module.Module

This head is useful for dealing with Siamese models which have multiple towers. For an input of type (N * num_towers) x C, this head can convert the output to N x (num_towers * C).

This head is used in case of PIRL https://arxiv.org/abs/1912.01991 and Jigsaw https://arxiv.org/abs/1603.09246 approaches.

__init__(*model_config:* vissl.utils.hydra_config.AttrDict, *num_towers:* *int*)

Parameters

- **model_config** (*AttrDict*) – dictionary config.MODEL in the config file
- **num_towers** (*int*) – number of towers in siamese model

forward(*batch:* *torch.Tensor*)

Parameters **batch** (*torch.Tensor*) – 2D torch tensor *(N * num_towers) x C* or 4D tensor of shape *(N * num_towers) x C x 1 x 1*

Returns *out (torch.Tensor)* – 2D output torch tensor *N x (C * num_towers)*

class vissl.models.heads.**SwAVPrototypesHead**(*model_config: vissl.utils.hydra_config.AttrDict, dims: List[int], use_bn: bool, num_clusters: int, use_bias: bool = True, return_embeddings: bool = True, skip_last_bn: bool = True, normalize_feats: bool = True*)

Bases: torch.nn.modules.module.Module

SwAV head used in https://arxiv.org/pdf/2006.09882.pdf paper.

The head is composed of 2 parts

1) projection of features to lower dimension like 128
2) feature classification into clusters (also called prototypes)

The projected features are L2 normalized before clustering step.

Input: 4D torch.tensor of shape (N x C x H x W)

Output: List(2D torch.tensor of shape N x num_clusters)

__init__(*model_config: vissl.utils.hydra_config.AttrDict, dims: List[int], use_bn: bool, num_clusters: int, use_bias: bool = True, return_embeddings: bool = True, skip_last_bn: bool = True, normalize_feats: bool = True*)

Parameters

- **model_config** (*AttrDict*) – dictionary config.MODEL in the config file
- **dims** (*int*) – dimensions of the linear layer. Must have length at least 2. Example: [2048, 2048, 128] attaches linear layer

  Linear(2048, 2048) -> BN -> Relu -> Linear(2048, 128)
- **use_bn** (*bool*) – whether to attach BatchNorm after Linear layer
- **num_clusters** (*List(int)*) – number of prototypes or clusters. Typically 3000. Example dims=[3000] will attach 1 prototype head.

  dims=[3000, 3000] will attach 2 prototype heads
- **use_bias** (*bool*) – whether the Linear layer should have bias or not
- **return_embeddings** (*bool*) – whether return the projected embeddings or not
- **skip_last_bn** (*bool*) – whether to attach BN + Relu at the end of projection head. .. rubric:: Example

  [2048, 2048, 128] with skip_last_bn=True attaches linear layer Linear(2048, 2048) -> BN -> Relu -> Linear(2048, 128)

  [2048, 2048, 128] with skip_last_bn=False attaches linear layer Linear(2048, 2048) -> BN -> Relu -> Linear(2048, 128) -> BN -> ReLU

  This could be particularly useful when performing full finetuning on hidden layers.

forward(*batch: torch.Tensor*)

Parameters **batch** (*4D torch.tensor*) – shape (N x C x H x W)

Returns List(2D torch.tensor of shape N x num_clusters)

61.4.3 vissl.models.trunks module

vissl.models.trunks.**register_model_trunk**(*name: str*)
Registers Self-Supervision Model Trunks.

This decorator allows VISSL to add custom model trunk, even if the model trunk itself is not part of VISSL. To use it, apply this decorator to a model trunk class, like this:

```
@register_model_trunk('my_model_trunk_name')
def my_model_trunk():
    ...
```

To get a model trunk from a configuration file, see *get_model_trunk()*.

vissl.models.trunks.**get_model_trunk**(*name: str*)
Given the model trunk name, construct the trunk if it's registered with VISSL.

61.5 vissl.losses package

61.5.1 vissl.losses.simclr_info_nce_loss

class vissl.losses.simclr_info_nce_loss.**SimclrInfoNCELoss**(*loss_config:* vissl.utils.hydra_config.AttrDict, *device: str = 'gpu'*)
Bases: classy_vision.losses.classy_loss.ClassyLoss

This is the loss which was proposed in SimCLR https://arxiv.org/abs/2002.05709 paper. See the paper for the details on the loss.

Config params: temperature (float): the temperature to be applied on the logits buffer_params:

world_size (int): total number of trainers in training embedding_dim (int): output dimensions of the features projects effective_batch_size (int): total batch size used (includes positives)

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)
Instantiates SimclrInfoNCELoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns SimclrInfoNCELoss instance.

forward(*output*, *target*)

class vissl.losses.simclr_info_nce_loss.**SimclrInfoNCECriterion**(*buffer_params*, *temperature: float*)
Bases: torch.nn.modules.module.Module

The criterion corresponding to the SimCLR loss as defined in the paper https://arxiv.org/abs/2002.05709.

Parameters

- **temperature** (*float*) – the temperature to be applied on the logits
- **buffer_params** – world_size (int): total number of trainers in training embedding_dim (int): output dimensions of the features projects effective_batch_size (int): total batch size used (includes positives)

precompute_pos_neg_mask()
We precompute the positive and negative masks to speed up the loss calculation

forward(*embedding: torch.Tensor*)
Calculate the loss. Operates on embeddings tensor.

static gather_embeddings(*embedding: torch.Tensor*)
Do a gather over all embeddings, so we can compute the loss. Final shape is like: (batch_size * num_gpus) x embedding_dim

61.5.2 vissl.losses.multicrop_simclr_info_nce_loss

class vissl.losses.multicrop_simclr_info_nce_loss.**MultiCropSimclrInfoNCELoss**(*loss_config:* vissl.utils.hydra_con *device: str = 'gpu'*)

Bases: *vissl.losses.simclr_info_nce_loss.SimclrInfoNCELoss*

Expanded version of the SimCLR loss. The SimCLR loss works only on 2 positives. We expand the loss to work for more positives following the multi-crop augmentation proposed in SwAV paper. See SwAV paper https://arxiv.org/abs/2006.09882 for the multi-crop augmentation details.

Config params: temperature (float): the temperature to be applied on the logits num_crops (int): number of positives used buffer_params:

world_size (int): total number of trainers in training embedding_dim (int): output dimensions of the features projects effective_batch_size (int): total batch size used (includes positives)

class vissl.losses.multicrop_simclr_info_nce_loss.**MultiCropSimclrInfoNCECriterion**(*buffer_paran temperature: float, num_crops: int*)

Bases: *vissl.losses.simclr_info_nce_loss.SimclrInfoNCECriterion*

The criterion corresponding to the expandion SimCLR loss (as defined in the paper https://arxiv.org/abs/2002.05709) using the multi-crop augmentaion proposed in SwAV paper. The multi-crop augmentation allows using more positives per image.

Parameters

- **temperature** (*float*) – the temperature to be applied on the logits
- **num_crops** (*int*) – number of positives
- **buffer_params** – world_size (int): total number of trainers in training embedding_dim (int): output dimensions of the features projects effective_batch_size (int): total batch size used (includes positives)

precompute_pos_neg_mask()
We precompute the positive and negative masks to speed up the loss calculation

forward(*embedding: torch.Tensor*)
Calculate the loss. Operates on embeddings tensor.

61.5.3 vissl.losses.swav_loss

class vissl.losses.swav_loss.**SwAVLoss**(*loss_config:* vissl.utils.hydra_config.AttrDict)
Bases: classy_vision.losses.classy_loss.ClassyLoss

This loss is proposed by the SwAV paper https://arxiv.org/abs/2006.09882 by Caron et al. See the paper for more details about the loss.

Config params: embedding_dim (int): the projection head output dimension temperature (float): temperature to be applied to the logits use_double_precision (bool): whether to use double precision for the loss.

This could be a good idea to avoid NaNs.

normalize_last_layer (bool): whether to normalize the last layer num_iters (int): number of sinkhorn algorithm iterations to make epsilon (float): see the paper for details num_crops (int): number of crops used crops_for_assign (List[int]): what crops to use for assignment num_prototypes (List[int]): number of prototypes temp_hard_assignment_iters (int): whether to do hard assignment for the initial

few iterations

output_dir (str): for dumping the debugging info in case loss becomes NaN

queue: queue_length (int): number of features to store and used in the scores start_iter (int): when to start using the queue for the scores local_queue_length (int): length of queue per gpu

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)
Instantiates SwAVLoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns SwAVLoss instance.

forward(*output: torch.Tensor*, *target: torch.Tensor*)

class vissl.losses.swav_loss.**SwAVCriterion**(*temperature: float*, *crops_for_assign: List[int]*, *num_crops: int*, *num_iters: int*, *epsilon: float*, *use_double_prec: bool*, *num_prototypes: List[int]*, *local_queue_length: int*, *embedding_dim: int*, *temp_hard_assignment_iters: int*, *output_dir: str*)
Bases: torch.nn.modules.module.Module

This criterion is used by the SwAV paper https://arxiv.org/abs/2006.09882 by Caron et al. See the paper for more details about the loss.

Config params: embedding_dim (int): the projection head output dimension temperature (float): temperature to be applied to the logits

num_iters (int): number of sinkhorn algorithm iterations to make epsilon (float): see the paper for details num_crops (int): number of crops used crops_for_assign (List[int]): what crops to use for assignment num_prototypes (List[int]): number of prototypes temp_hard_assignment_iters (int): whether to do hard assignment for the initial

few iterations

output_dir (str): for dumping the debugging info in case loss becomes NaN

local_queue_length (int): length of queue per gpu

distributed_sinkhornknopp(*Q: torch.Tensor*)
Apply the distributed sinknorn optimization on the scores matrix to find the assignments

forward(*scores: torch.Tensor*, *head_id: int*)

update_emb_queue(*emb*)

compute_queue_scores(*head*)

initialize_queue()

61.5.4 vissl.losses.bce_logits_multiple_output_single_target

class vissl.losses.bce_logits_multiple_output_single_target.**BCELogitsMultipleOutputSingleTa**

Bases: classy_vision.losses.classy_loss.ClassyLoss

__init__(*loss_config:* vissl.utils.hydra_config.AttrDict)

Intializer for the sum cross-entropy loss. For a single tensor, this is equivalent to the cross-entropy loss. For a list of tensors, this computes the sum of the cross-entropy losses for each tensor in the list against the target.

Config params: reduction: specifies reduction to apply to the output, optional normalize_output: Whether to L2 normalize the outputs world_size: total number of gpus in training. automatically inferred by vissl

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)

Instantiates BCELogitsMultipleOutputSingleTargetLoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns BCELogitsMultipleOutputSingleTargetLoss instance.

forward(*output: Union[torch.Tensor, List[torch.Tensor]]*, *target: torch.Tensor*)

For each output and single target, loss is calculated. The returned loss value is the sum loss across all outputs.

61.5.5 vissl.losses.swav_momentum_loss

class vissl.losses.swav_momentum_loss.**SwAVMomentumLoss**(*loss_config:* vissl.utils.hydra_config.AttrDict)

Bases: classy_vision.losses.classy_loss.ClassyLoss

This loss extends the SwAV loss proposed in paper https://arxiv.org/abs/2006.09882 by Caron et al. The loss combines the benefits of using the SwAV approach with the momentum encoder as used in MoCo.

Config params: momentum (float): for the momentum encoder momentum_eval_mode_iter_start (int): from what iteration should the momentum encoder

network be in eval mode

embedding_dim (int): the projection head output dimension temperature (float): temperature to be applied to the logits use_double_precision (bool): whether to use double precision for the loss.

This could be a good idea to avoid NaNs.

normalize_last_layer (bool): whether to normalize the last layer num_iters (int): number of sinkhorn algorithm iterations to make epsilon (float): see the paper for details num_crops (int): number of crops used crops_for_assign (List[int]): what crops to use for assignment num_prototypes (List[int]): number of prototypes queue:

queue_length (int): number of features to store and used in the scores start_iter (int): when to start using the queue for the scores local_queue_length (int): length of queue per gpu

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)
Instantiates SwAVMomentumLoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns SwAVMomentumLoss instance.

initialize_queue()

load_state_dict(*state_dict*, **args*, ***kwargs*)
Restore the loss state given a checkpoint

Parameters **state_dict** (*serialized via torch.save*) –

forward(*output:* *torch.Tensor*, **args*, ***kwargs*)

distributed_sinkhornknopp(*Q:* *torch.Tensor*)
Apply the distributed sinknorn optimization on the scores matrix to find the assignments

update_emb_queue()

compute_queue_scores(*head*)

61.5.6 vissl.losses.moco_loss

class vissl.losses.moco_loss.**MoCoLossConfig**(*embedding_dim*, *queue_size*, *momentum*, *temperature*)
Bases: vissl.losses.moco_loss._MoCoLossConfig

Settings for the MoCo loss

static defaults() → *vissl.losses.moco_loss.MoCoLossConfig*

class vissl.losses.moco_loss.**MoCoLoss**(*config:* vissl.losses.moco_loss.MoCoLossConfig)
Bases: classy_vision.losses.classy_loss.ClassyLoss

This is the loss which was proposed in the "Momentum Contrast for Unsupervised Visual Representation Learning" paper, from Kaiming He et al. See http://arxiv.org/abs/1911.05722 for details and https://github.com/facebookresearch/moco for a reference implementation, reused here

Config params: embedding_dim (int): head output output dimension queue_size (int): number of elements in queue momentum (float): encoder momentum value for the update temperature (float): temperature to use on the logits

classmethod from_config(*config:* vissl.losses.moco_loss.MoCoLossConfig)
Instantiates MoCoLoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns MoCoLoss instance.

forward(*query:* *torch.Tensor*, **args*, ***kwargs*) → torch.Tensor
Given the encoder queries, the key and the queue of the previous queries, compute the cross entropy loss for this batch

Parameters **query** – output of the encoder given the current batch

Returns loss

load_state_dict(*state_dict*, **args*, ***kwargs*)
Restore the loss state given a checkpoint

Parameters **state_dict** (*serialized via torch.save*) –

61.5.7 vissl.losses.nce_loss

class vissl.losses.nce_loss.**NCELossWithMemory**(*loss_config:* vissl.utils.hydra_config.AttrDict)

Bases: classy_vision.losses.classy_loss.ClassyLoss

Distributed version of the NCE loss. It performs an "all_gather" to gather the allocated buffers like memory no a single gpu. For this, Pytorch distributed backend is used. If using NCCL, one must ensure that all the buffer are on GPU. This class supports training using both NCE and CrossEntropy (InfoNCE).

This loss is used by NPID (https://arxiv.org/pdf/1805.01978.pdf), NPID++ and PIRL (https://arxiv.org/abs/1912.01991) approaches.

Written by: Ishan Misra (imisra@fb.com)

Config params: norm_embedding (bool): whether to normalize embeddings temperature (float): the temperature to apply to logits norm_constant (int): Z parameter in the NCEAverage update_mem_with_emb_index (int): In case we have multiple embeddings used

in the nce loss, specify which embedding to use to update the memory.

loss_type (str): options are "nce" | "cross_entropy". Using the cross_entropy turns the loss into InfoNCE loss.

loss_weights (List[float]): if the NCE loss is computed between multiple pairs, we can set a loss weight per term can be used to weight different pair contributions differently

negative_sampling_params: num_negatives (int): how many negatives to contrast with type (str): how to select the negatives. options "random"

memory_params:

memory_size (int): number of training samples as all the samples are stored in memory

embedding_dim (int): the projection head output dimension momentum (int): momentum to use to update the memory norm_init (bool): whether to L2 normalize the initialized memory bank update_mem_on_forward (bool): whether to update memory on the forward pass

num_train_samples (int): number of unique samples in the training dataset

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)

Instantiates NCELossWithMemory from configuration.

Parameters **loss_config** – configuration for the loss

Returns NCELossWithMemory instance.

forward(*output: Union[torch.Tensor, List[torch.Tensor]]*, *target: torch.Tensor*)

For each output and single target, loss is calculated.

sync_memory()

Sync memory across all processes before first forward pass. Only needed in the distributed case. After the first forward pass, the update_memory function in NCEAverage does a gather over all embeddings, so memory stays in sync. Doing a gather over embeddings is O(batch size). Syncing memory is O(num items in memory). Generally, batch size << num items in memory. So, we prefer doing the syncs in update_memory.

update_memory(*embedding*, *y*)

class vissl.losses.nce_loss.**NCEAverage**(*memory_params*, *negative_sampling_params*, *T=0.07*, *Z=- 1*, *loss_type='nce'*)

Bases: torch.nn.modules.module.Module

Computes the scores of the model embeddings against the `positive' and `negative' samples from the Memory Bank. This class does *NOT* compute the actual loss, just the scores, i.e., inner products followed by normalizations/exponentiation etc.

forward(*embedding*, *y*, *idx=None*, *update_memory_on_forward=None*)

compute_partition_function(*out*)

do_negative_sampling(*embedding*, *y*, *num_negatives*)

setup_negative_sampling(*negative_sampling_params*)

init_memory(*memory_params*)

update_memory(*embedding*, *y*)

class vissl.losses.nce_loss.**AliasMethod**(*probs*)
Bases: torch.nn.modules.module.Module

A fast way to sample from a multinomial distribution. Faster than torch.multinomial or np.multinomial. The setup (__init__) for this class is slow, however `draw' (actual sampling) is fast.

draw(*N*)
Draw N samples from multinomial :param N: number of samples :return: samples

class vissl.losses.nce_loss.**NumpySampler**(*high*)
Bases: object

draw(*num_negatives*)

class vissl.losses.nce_loss.**NCECriterion**(*nLem*)
Bases: torch.nn.modules.module.Module

forward(*x*, *targets*)

61.5.8 vissl.losses.deepclusterv2_loss

class vissl.losses.deepclusterv2_loss.**DeepClusterV2Loss**(*loss_config:* vissl.utils.hydra_config.AttrDict)
Bases: classy_vision.losses.classy_loss.ClassyLoss

Loss used for DeepClusterV2 approach as provided in SwAV paper https://arxiv.org/abs/2006.09882

Config params: DROP_LAST (bool): automatically inferred from DATA.TRAIN.DROP_LAST BATCHSIZE_PER_REPLICA (int): 256 # automatically inferred from

DATA.TRAIN.BATCHSIZE_PER_REPLICA

num_crops (int): 2 # automatically inferred from DATA.TRAIN.TRANSFORMS temperature (float): 0.1 num_clusters (List[int]): [3000, 3000, 3000] kmeans_iters (int): 10 crops_for_mb: [0] embedding_dim: 128 num_train_samples (int): -1 # @auto-filled

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)
Instantiates DeepClusterV2Loss from configuration.

Parameters **loss_config** – configuration for the loss

Returns DeepClusterV2Loss instance.

forward(*output: torch.Tensor*, *idx: int*)

init_memory(*dataloader*, *model*)

update_memory_bank(*emb*, *idx*)

cluster_memory()

61.5.9 vissl.losses.cross_entropy_multiple_output_single_target

class vissl.losses.cross_entropy_multiple_output_single_target.**CrossEntropyMultipleOutputS**

Bases: classy_vision.losses.classy_loss.ClassyLoss

Intializer for the sum cross-entropy loss. For a single tensor, this is equivalent to the cross-entropy loss. For a list of tensors, this computes the sum of the cross-entropy losses for each tensor in the list against the target.

Config params: weight: weight of sample, optional ignore_index: sample should be ignored for loss, optional reduction: specifies reduction to apply to the output, optional temperature: specify temperature for softmax. Default 1.0

classmethod from_config(*loss_config:* vissl.utils.hydra_config.AttrDict)
Instantiates CrossEntropyMultipleOutputSingleTargetLoss from configuration.

Parameters **loss_config** – configuration for the loss

Returns CrossEntropyMultipleOutputSingleTargetLoss instance.

forward(*output: Union[torch.Tensor, List[torch.Tensor]]*, *target: torch.Tensor*)
For each output and single target, loss is calculated. The returned loss value is the sum loss across all outputs.

61.6 vissl.hooks package

class vissl.hooks.**SSLClassyHookFunctions**(*value*)
Bases: enum.Enum

Enumeration of all the hook functions in the ClassyHook class.

on_start = 1

on_phase_start = 2

on_forward = 3

on_loss_and_meter = 4

on_backward = 5

on_update = 6

on_step = 7

on_phase_end = 8

on_end = 9

vissl.hooks.**default_hook_generator**(*cfg:* vissl.utils.hydra_config.AttrDict) → List[classy_vision.hooks.classy_hook.ClassyHook]
The utility function that prepares all the hoooks that will be used in training based on user selection. Some basic hooks are used by default.

Optional hooks:

- Tensorboard hook,
- loss specific hooks (swav loss, deepcluster loss, moco loss) used only when the loss is being used

- model complexity hook (if user wants to compute model flops, activations, params) enable the hook via MODEL.MODEL_COMPLEXITY.COMPUTE_COMPLEXITY = True

Returns *hooks (List(functions))* – list containing the hook functions that will be used

61.6.1 vissl.hooks.deepclusterv2_hooks module

class vissl.hooks.deepclusterv2_hooks.**InitMemoryHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Initialize the memory banks. Valid only for DeepClusterV2 training

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_start(*task*) → None
At the begining of the training, initialize the memory banks

class vissl.hooks.deepclusterv2_hooks.**ClusterMemoryHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Cluster the memory banks with distributed k-means. Valid only for DeepClusterV2 trainings.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(*task*) → None
At the beginning of each epochs, cluster the memory banks with distributed k-means

61.6.2 vissl.hooks.log_hooks module

All the hooks involved in human-readable logging

class vissl.hooks.log_hooks.**LogGpuStatsHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Hook executed at the start of training and after every training iteration is done. Logs Gpu nvidia-smi stats to logger streams: at the start of training and after 50 training iterations.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_start(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Logs Gpu nvidia-smi stats to logger streams.

on_step(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Print the nvidia-smi stats again to get more accurate nvidia-smi useful for monitoring memory usage.

class vissl.hooks.log_hooks.**LogLossLrEtaHook**(*btime_freq: Optional[int] = None*)
Bases: classy_vision.hooks.classy_hook.ClassyHook

Hook executed after every parameters update step. Logs training stats like: LR, iteration, ETA, batch time etc to logger streams.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

__init__(*btime_freq: Optional[int] = None*) → None

Parameters **btime_freq** – if specified, logs average batch time of rolling_freq batches also.

on_update(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Executed after after parameter update. If the current phase is training, and it's a logging iteration, we compute and log several helpul training stats to keep track of ongoing training.

For monitoring the batch size (average training iteration time), we allow monitoring the stats (optionally) for every N iterations to get better idea about the batch time and training eta.

Set the btime_freq input using cfg.PERF_STAT_FREQUENCY=N ensuring that cfg.MONITOR_PERF_STATS = True.

class vissl.hooks.log_hooks.**LogLossMetricsCheckpointHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Hook called after every forward pass (to check training doesn't give NaN), after every step and at the end of epoch (to check if the model should be checkpointed) and print the meters values at the end of every phase.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time a model forward is done and make sure that the model forward output is not NaN. If we encounter NaN as the model output, we checkpoint the model to enable debugging and also checkpoint the model input sample, model output.

on_step(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
In some cases, we might want to checkpoint after certain number of iterations. If we want to checkpoint after every N iterations, check the checkpoint frequency matches and checkpoint if it does.

on_phase_end(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called at the end of each phase and forward. We log the metrics and also save the checkpoint. We pass the mode: phase or iteration

class vissl.hooks.log_hooks.**LogPerfTimeMetricsHook**(*log_freq: Optional[int] = None*)
Bases: classy_vision.hooks.classy_hook.ClassyHook

Computes and prints performance metrics. Logs at the end of a phase or every log_freq if specified by user.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

__init__(*log_freq: Optional[int] = None*) → None

Parameters **log_freq** – if specified, logs every log_freq batches also.

on_phase_start(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Initialize start time and reset perf stats

on_loss_and_meter(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Log performance metrics every log_freq batches, if log_freq is not None.

on_phase_end(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Log performance metrics at the end of a phase if log_freq is None.

61.6.3 vissl.hooks.moco_hooks module

class vissl.hooks.moco_hooks.**MoCoHook**(*momentum: float*, *shuffle_batch: bool = True*)
Bases: classy_vision.hooks.classy_hook.ClassyHook

This hook corresponds to the loss proposed in the "Momentum Contrast for Unsupervised Visual Representation Learning" paper, from Kaiming He et al. See http://arxiv.org/abs/1911.05722 for details and https://github.com/facebookresearch/moco for a reference implementation, reused here.

Called after every forward pass to update the momentum encoder. At the beginning of training i.e. after 1st forward call, the encoder is contructed and updated.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None

- Update the momentum encoder.
- Compute the key reusing the updated moco-encoder. If we use the batch shuffling, the perform global shuffling of the batch and then run the moco encoder to compute the features. We unshuffle the computer features and use the features as "key" in computing the moco loss.

61.6.4 vissl.hooks.state_update_hooks module

class vissl.hooks.state_update_hooks.**SSLModelComplexityHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Logs the number of paramaters, forward pass FLOPs and activations of the model. Adapted from: https://github.com/facebookresearch/ClassyVision/blob/master/classy_vision/hooks/model_complexity_hook.py#L20 # NOQA

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_start(*task*) → None
Before the training starts, run one forward only pass of the model on the dummy input of shape specified by user in MODEL.MODEL_COMPLEXITY.INPUT_SHAPE We calculate the flops, activations and number of params in the model.

class vissl.hooks.state_update_hooks.**SetDataSamplerEpochHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

We use DistributedDataSampler for sampling the data. At the beginnning of each training epoch/phase, we need to set the epoch for the sampler.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called at the start of each epoch or phase to set the data sampler epoch. This is important to ensure the data is shuffled and the shuffling can be reproduced deterministically if the training is resumed from a checkpoint.

class vissl.hooks.state_update_hooks.**UpdateBatchesSeenHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Book-keeping only hook. Tracks how many forward passes have been done. aka how many batches have been seen by the trainer irrespective of the train or test phase. updates task.batches

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time forward pass is triggered. We update the number of batches we have seen. This is useful for debugging.

class vissl.hooks.state_update_hooks.**UpdateTrainIterationNumHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Book-keeping hook: updates the training iteration number (only updated if it's a training phase). task.iteration is updated.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time forward pass is triggered. We update the number of batches we have seen. This is useful for debugging.

class vissl.hooks.state_update_hooks.**UpdateTrainBatchTimeHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

After after parameters update step (training phase), we update the batch time aka the training time for the current iteration.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time forward pass is triggered. We update the number of batches we have seen. This is useful for debugging.

class vissl.hooks.state_update_hooks.**UpdateTestBatchTimeHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Include the batch time for test phase as well and called every time loss has been computed. Only updates task.batch_time if it's a test phase and train phase is already updated by UpdateTrainBatchTimeHook hook.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time a loss has been computed. Append the batch time for test phase.

class vissl.hooks.state_update_hooks.**CheckNanLossHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

After every loss computation, verify the loss is not infinite. Called for both training/test phase.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called each time a loss has been computed and checks that loss is not None.

class vissl.hooks.state_update_hooks.**FreezeParametersHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Hook that helps to freeze some specified model parameters for certain number of training iterations. The parameters are specified in a dictionary containing {param_name: frozen_iterations}. Used in SwAV training.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
After every backward pass and before updating the parameters, check if there are parameters that should stay frozen. Set the grad to None for those params.

61.6.5 vissl.hooks.swav_hooks module

class vissl.hooks.swav_hooks.**SwAVUpdateQueueScoresHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

Update queue scores, useful with small batches and helps getting meaningful gradients.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(*task*) → None
If we want to use queue in SwAV training, update the queue scores after every forward.

class vissl.hooks.swav_hooks.**NormalizePrototypesHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

L2 Normalize the prototypes in swav training. Optional.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Optionally normalize prototypes

61.6.6 vissl.hooks.swav_momentum_hooks module

class vissl.hooks.swav_momentum_hooks.**SwAVMomentumHook**(*momentum: float*, *momentum_eval_mode_iter_start: int*, *crops_for_assign: List[int]*)
Bases: classy_vision.hooks.classy_hook.ClassyHook

This hook is for the extension of the SwAV loss proposed in paper https://arxiv.org/abs/2006.09882 by Caron et al. The loss combines the benefits of using the SwAV approach with the momentum encoder as used in MoCo.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

__init__(*momentum: float*, *momentum_eval_mode_iter_start: int*, *crops_for_assign: List[int]*)

Parameters

- **momentum** (*float*) – for the momentum encoder
- **momentum_eval_mode_iter_start** (*int*) – from what iteration should the momentum encoder network be in eval mode
- **crops_for_assign** (*List[int]*) – what crops to use for assignment

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Forward pass with momentum network. We forward momentum encoder only on the single resolution crops that are used for assignment in the swav loss.

class vissl.hooks.swav_momentum_hooks.**SwAVMomentumNormalizePrototypesHook**
Bases: classy_vision.hooks.classy_hook.ClassyHook

L2 Normalize the prototypes in swav training. Optional. We normalize the momentum_encoder output prototypes as well additionally.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_forward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_phase_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_update(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Optionally normalize prototypes

61.6.7 vissl.hooks.tensorboard_hooks module

class vissl.hooks.tensorboard_hook.**SSLTensorboardHook**(*tb_writer: torch.utils.tensorboard.writer.SummaryWriter*, *log_params: bool = False*, *log_params_every_n_iterations: int = - 1*, *log_params_gradients: bool = False*)
Bases: classy_vision.hooks.classy_hook.ClassyHook

SSL Specific variant of the Classy Vision tensorboard hook

on_loss_and_meter(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_backward(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_start(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_end(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

on_step(**args*, ***kwargs*) → None
Derived classes can set their hook functions to this.

This is useful if they want those hook functions to not do anything.

__init__(*tb_writer: torch.utils.tensorboard.writer.SummaryWriter*, *log_params: bool = False*, *log_params_every_n_iterations: int = - 1*, *log_params_gradients: bool = False*) → None
The constructor method of SSLTensorboardHook.

Parameters

- **tb_writer** – Tensorboard SummaryWriter instance

- **log_params** (*bool*) – whether to log model params to tensorboard
- **log_params_every_n_iterations** (*int*) – frequency at which parameters should be logged to tensorboard
- **log_params_gradients** (*bool*) – whether to log params gradients as well to tensorboard.

on_forward(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called after every forward if tensorboard hook is enabled. Logs the model parameters if the training iteration matches the logging frequency.

on_phase_start(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called at the start of every epoch if the tensorboard hook is enabled. Logs the model parameters once at the beginning of training only.

on_phase_end(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called at the end of every epoch if the tensorboard hook is enabled. Log model parameters and/or parameter gradients as set by user in the tensorboard configuration. Also resents the CUDA memory counter.

on_update(*task: classy_vision.tasks.classy_task.ClassyTask*) → None
Called after every parameters update if tensorboard hook is enabled. Logs the parameter gradients if they are being set to log, log the scalars like training loss, learning rate, average training iteration time, batch size per gpu, img/sec/gpu, ETA, gpu memory used, peak gpu memory used.

61.7 vissl.optimizers package

vissl.optimizers.**get_optimizer_param_groups**(*model*, *model_config*, *optimizer_config*, *optimizer_schedulers*)
Go through all the layers, sort out which parameters should be regularized, unregularized and optimization settings for the head/trunk. We filter the trainable params only and add them to the param_groups.

Returns

param_groups (List[Dict]) –

[

{ “params”: trunk_regularized_params, “lr”: lr_value, “weight_decay”: wd_value,

}, {

”params”: trunk_unregularized_params, “lr”: lr_value, “weight_decay”: 0.0,

}, {

”params”: head_regularized_params, “lr”: head_lr_value, “weight_decay”: head_weight_decay,

}, {

”params”: head_unregularized_params, “lr”: head_lr_value, “weight_decay”: 0.0,

}, {

”params”: remaining_regularized_params, “lr”: lr_value

}

]

61.7.1 vissl.optimizers.optimizer_helper module

vissl.optimizers.optimizer_helper.**get_optimizer_param_groups**(*model*, *model_config*, *optimizer_config*, *optimizer_schedulers*)

Go through all the layers, sort out which parameters should be regularized, unregularized and optimization settings for the head/trunk. We filter the trainable params only and add them to the param_groups.

Returns

param_groups (List[Dict]) –

[

{ "params": trunk_regularized_params, "lr": lr_value, "weight_decay": wd_value,

}, {

"params": trunk_unregularized_params, "lr": lr_value, "weight_decay": 0.0,

}, {

"params": head_regularized_params, "lr": head_lr_value, "weight_decay": head_weight_decay,

}, {

"params": head_unregularized_params, "lr": head_lr_value, "weight_decay": 0.0,

}, {

"params": remaining_regularized_params, "lr": lr_value

}

]

61.7.2 vissl.optimizers.param_scheduler.cosine_warm_restart_scheduler module

class vissl.optimizers.param_scheduler.cosine_warm_restart_scheduler.**CosineWaveTypes**(*value*)

Bases: str, enum.Enum

An enumeration.

half = 'half'

full = 'full'

class vissl.optimizers.param_scheduler.cosine_warm_restart_scheduler.**CosineWarmRestartSched**

Bases: classy_vision.optim.param_scheduler.classy_vision_param_scheduler.ClassyParamScheduler

Changes the param value after every epoch based on a cosine schedule. The schedule is updated after every train step by default.

Can be used for cosine learning rate with warm restarts. For restarts, we calculate what will be the maximum learning rate after every restart. There are 3 options:

- Option 1: LR after every restart is same as original max LR
- Option 2: LR after every restart decays with a fixed LR multiplier
- **Option 3: LR after every restart is adaptively calculated such that the resulting** max LR matches the original cosine wave LR.

Parameters

- **wave_type** – half | full
- **lr_multiplier** – float value -> LR after every restart decays with a fixed LR multiplier
- **is_adaptive** – True -> if after every restart, maximum LR is adaptively calculated such that the resulting max LR matches the original cosine wave LR.
- **update_interval** – step | epoch -> if the LR should be updated after every training iteration or after training epoch

Example

```
start_value: 0.1
end_value: 0.0001
restart_interval_length: 0.5  # for 1 restart
wave_type: half
lr_multiplier: 1.0  # for using a decayed max LR value at every restart
use_adaptive_decay: False
```

classmethod from_config(*config: Dict[str, Any]*) → *vissl.optimizers.param_scheduler.cosine_warm_restart_scheduler.*
Instantiates a CosineWarmRestartScheduler from a configuration.

Parameters **config** – A configuration for a CosineWarmRestartScheduler. See __init__() for parameters expected in the config.

Returns A CosineWarmRestartScheduler instance.

61.7.3 vissl.optimizers.param_scheduler.inverse_sqrt_decay module

class vissl.optimizers.param_scheduler.inverse_sqrt_decay.**InverseSqrtScheduler**(*start_value: float, warmup_interval float, update_interval: classy_vision.opt = <UpdateInterval.STEP: 'step'>*)

Bases: classy_vision.optim.param_scheduler.classy_vision_param_scheduler.ClassyParamScheduler

Decay the LR based on the inverse square root of the update number.

Example

```
start_value: 4.8
warmup_interval_length: 0.1
```

Corresponds to a inverse sqrt decay schedule with values in [4.8, 0]

classmethod from_config(*config: Dict[str, Any]*) → *vissl.optimizers.param_scheduler.inverse_sqrt_decay.InverseSqrtS*
Instantiates a InverseSqrtScheduler from a configuration.

Parameters **config** – A configuration for a InverseSqrtScheduler. See __init__() for parameters expected in the config.

Returns A InverseSqrtScheduler instance.

61.8 vissl.trainer package

vissl.trainer.trainer_main.**build_task**(*config*)

Builds a ClassyTask from a config.

This assumes a 'name' key in the config which is used to determine what task class to instantiate. For instance, a config *{"name": "my_task", "foo": "bar"}* will find a class that was registered as "my_task" (see `register_task()`) and call .from_config on it.

class vissl.trainer.trainer_main.**SelfSupervisionTrainer**(*cfg:* vissl.utils.hydra_config.AttrDict, *dist_run_id:* *str*, *checkpoint_path:* *str* = *None*, *checkpoint_folder:* *str* = *None*, *hooks:* *List[classy_vision.hooks.classy_hook.ClassyHook]* = *None*)

Bases: `object`

The main entry point for any training or feature extraction workflows in VISSL.

The trainer constructs a train_task which prepares all the components of the training (optimizer, loss, meters, model etc) using the settings specified by user in the yaml config file. See the vissl/trainer/train_task.py for more details.

Parameters

- **cfg** (*AttrDict*) – user specified input config that has optimizer, loss, meters etc settings relevant to the training
- **dist_run_id** (*str*) – For multi-gpu training with PyTorch, we have to specify how the gpus are going to rendezvous. This requires specifying the communication method: file, tcp and the unique rendezvous run_id that is specific to 1 run. We recommend:

  1) for 1node: use init_method=tcp and run_id=auto

  2) for multi-node, use init_method=tcp and specify run_id={master_node}:{port}
- **checkpoint_path** (*str*) – if the training is being resumed from a checkpoint, path to the checkpoint. The tools/run_distributed_engines.py automatically looks for the checkpoint in the checkpoint directory.
- **checkpoint_folder** (*str*) – what directory to use for checkpointing. The tools/run_distributed_engines.py creates the directory based on user input in the yaml config file.
- **hooks** (*List[ClassyHooks]*) – the list of hooks to use during the training. The hooks vissl/engines/{train, extract_features}.py determine the hooks.

setup_distributed(*use_gpu: bool*)

Setup the distributed training. VISSL support both GPU and CPU only training.

(1) Initialize the torch.distributed.init_process_group if the distributed is not already initialized. The init_method, backend are specified by user in the yaml config file. See vissl/defaults.yaml file for description on how to set init_method, backend.

(2) We also set the global cuda device index using torch.cuda.set_device or cpu device

train()

The train workflow. We get the training loop to use (vissl default is standard_train_step) but the user can create their own training loop and specify the name TRAINER.TRAIN_STEP_NAME

The training happens: 1. Execute any hooks at the start of training (mostly resets the variable like

iteration num phase_num etc)

2. For each epoch (train or test), run the hooks at the start of an epoch. Mostly involves setting things like timer, setting dataloader epoch etc
3. Execute the training loop (1 training iteration) involving forward, loss, backward, optimizer update, metrics collection etc.
4. At the end of epoch, sync meters and execute hooks at the end of phase. Involves things like checkpointing model, logging timers, logging to tensorboard etc

extract()

Extract workflow supports multi-gpu feature extraction. Since we are only extracting features, only the model is built (and initialized from some model weights file if specified by user). The model is set to the eval mode fully.

The features are extracted for whatever data splits (train, val, test) etc that user wants.

61.8.1 vissl.train_task package

class vissl.trainer.train_task.**SelfSupervisionTask**(*config:* vissl.utils.hydra_config.AttrDict)

Bases: classy_vision.tasks.classification_task.ClassificationTask

A task prepares and holds all the components of a training like optimizer, datasets, dataloaders, losses, meters etc. Task also contains the variable like training iteration, epoch number etc. that are updated during the training.

We prepare every single component according to the parameter settings user wants and specified in the yaml config file.

Task also supports 2 additional things: 1) converts the model BatchNorm layers to the synchronized batchnorm 2) sets mixed precision (apex and pytorch both supported)

set_device()

Set the training device: whether gpu or cpu. We use the self.device in the rest of the workflow to determine if we should do cpu only training or use gpu. set MACHINE.DEVICE = "gpu" or "cpu"

set_ddp_bucket_cap_mb()

PyTorch DDP supports setting the bucket_cap_mb for all reduce. Tuning this parameter can help with the speed of the model. We use the default pytorch value of 25MB.

set_available_splits()

Given the data settings, we determine if we are using both train and test datasets. If TEST_MODEL=true, we will add the test to the available_splits. If TEST_ONLY=false, we add train to the split as well.

set_amp_args()

Two automatic mixed precision implementations are available: Apex's and PyTorch's.

- If Apex's AMP is enabled, amp_args is a dictionary containing arguments

to be passed to amp.initialize. Set to None to disable amp. To enable mixed precision training, pass amp_args={"opt_level": "O1"} here. See https://nvidia.github.io/apex/amp.html for more info.

- If Pytorch's AMP is enabled, no arguments are needed.

set_checkpoint_path(*checkpoint_path: str*)

Set the checkpoint path for the training

set_checkpoint_folder(*checkpoint_folder: str*)
Set the checkpoint folder for the training

set_iteration(*iteration*)
Set the iteration number. we maintain and store the iteration in the state itself. It counts total number of iterations we do in training phases. Updated after every forward pass of training step in UpdateTrainIterationNumHook. Starts from 1

classmethod from_config(*config*)
Create the task from the yaml config input.

get_config()
Utility function to store and use the config that was used for the given training.

build_datasets()
Get the datasets for the data splits we will use in the training. The set_available_splits variable determines the splits used in the training.

build_dataloaders(*pin_memory: bool*) → torch.utils.data.dataloader.DataLoader
Build PyTorch dataloaders for all the available_splits. We construct the standard PyTorch Dataloader and allow setting all dataloader options.

get_global_batchsize()
Return global batchsize used in the training across all the trainers. We check what phase we are in (train or test) and get the dataset used in that phase. We call get_global_batchsize() of the dataset.

recreate_data_iterator(*phase_type*, *epoch*, *compute_start_iter*)
Recreate data iterator (including multiprocessing workers) and destroy the previous iterators.

This is called when we load a new checkpoint or when phase changes during the training (one epoch to the next). DataSampler may need to be informed on those events to update the epoch and start_iteration so that the data is deterministically shuffled, so we call them here.

run_hooks(*hook_function_name*)
Override the ClassyTask run_hook function and run the hooks whenever called

prepare_optimizer()
Constructs the optimizer using the user defined settings in the yaml config. The model must be on the correct device (cuda or cpu) by this point.

prepare(*pin_memory: bool = False*)
Prepares the task: - dataloaders - model - copy model to correct device - meters - loss - optimizer - LR schedulers - AMP state - resume from a checkpoint if available

prepare_extraction(*pin_memory: bool = False*)
Prepares a light-weight task for feature extraction on multi-gpu. The model runs in eval mode only.

property enable_manual_gradient_reduction
Lazily initial the enable flag once when model is not None.

set_manual_gradient_reduction() → None
Called during __init__ to set a flag if manual gradient reduction is enabled.

61.8.2 vissl.trainer.train_steps module

Here we create all the custom train steps required for SSL model trainings.

vissl.trainer.train_steps.**register_train_step**(*name*)
Registers Self-Supervision Train step.

This decorator allows VISSL to add custom train steps, even if the train step itself is not part of VISSL. To use it, apply this decorator to a train step function, like this:

```
@register_train_step('my_step_name')
def my_step_name():
    ...
```

To get a train step from a configuration file, see *get_train_step()*.

vissl.trainer.train_steps.**get_train_step**(*train_step_name: str*)
Lookup the train_step_name in the train step registry and return. If the train step is not implemented, asserts will be thrown and workflow will exit.

61.8.3 vissl.trainer.train_steps.standard_train_step module

This is the train step that"s most commonly used in most of the model trainings.

vissl.trainer.train_steps.standard_train_step.**construct_sample_for_model**(*batch_data*, *task*)
Given the input batch from the dataloader, verify the input is as expected: the input data and target data is present in the batch. In case of multi-input trainings like PIRL, make sure the data is in right format i.e. the multiple input should be nested under a common key "input".

vissl.trainer.train_steps.standard_train_step.**standard_train_step**(*task*)
Single training iteration loop of the model.

Performs: data read, forward, loss computation, backward, optimizer step, parameter updates.

Various intermediate steps are also performed: - logging the training loss, training eta, LR, etc to loggers - logging to tensorboard, - performing any self-supervised method specific operations (like in MoCo approach, the momentum encoder is updated), computing the scores in swav - checkpointing model if user wants to checkpoint in the middle of an epoch

61.9 vissl.utils package

61.9.1 vissl.utils.instance_retrieval_utils.data_util module

vissl.utils.instance_retrieval_utils.data_util.**is_revisited_dataset**(*dataset_name: str*)
Computes whether the specified dataseet name is a revisited version of the oxford and paris datasets. simply looks for pattern "roxford5k" and "rparis6k" in specified dataset_name.

vissl.utils.instance_retrieval_utils.data_util.**is_instre_dataset**(*dataset_name: str*)
Returns True if the dataset name is "instre". Helper function used in code at several places.

vissl.utils.instance_retrieval_utils.data_util.**is_whiten_dataset**(*dataset_name: str*)
Returns if the dataset specified has name "whitening". User can use any dataset they want for whitening.

vissl.utils.instance_retrieval_utils.data_util.**add_bias_channel**(*x*, *dim: int = 1*)

Adds a bias channel useful during pooling + whitening operation.

vissl.utils.instance_retrieval_utils.data_util.**flatten**(*x: torch.Tensor*, *keepdims: bool = False*)

Flattens B C H W input to B C*H*W output, optionally retains trailing dimensions.

vissl.utils.instance_retrieval_utils.data_util.**gem**(*x: torch.Tensor*, *p: int = 3*, *eps: float = 1e-06*, *clamp: bool = True*, *add_bias: bool = False*, *keepdims: bool = False*)

Gem pooling on the given tensor.

Parameters

- **x** (*torch.Tensor*) – tensor on which the pooling should be done
- **p** (*int*) – pooling number. If p=inf then simply perform max_pool2d If p=1 and x tensor has grad, simply perform avg_pool2d else, perform Gem pooling for specified p
- **eps** (*float*) – if clamping the x tensor, use the eps for clamping
- **clamp** (*float*) – whether to clamp the tensor
- **add_bias** (*bool*) – whether to add the biad channel
- **keepdims** (*bool*) – whether to flatten or keep the dimensions as is

Returns *x (torch.Tensor)* – Gem pooled tensor

vissl.utils.instance_retrieval_utils.data_util.**l2n**(*x: torch.Tensor*, *eps: float = 1e-06*, *dim: int = 1*)

L2 normalize the input tensor along the specified dimension

Parameters

- **x** (*torch.Tensor*) – the tensor to normalize
- **eps** (*float*) – epsilon to use to normalize to avoid the inf output
- **dim** (*int*) – along which dimension to L2 normalize

Returns *x (torch.Tensor)* – L2 normalized tensor

class vissl.utils.instance_retrieval_utils.data_util.**MultigrainResize**(*size: int*, *largest: bool = False*, ***kwargs*)

Bases: torchvision.transforms.transforms.Resize

Resize with a *largest=False* argument allowing to resize to a common largest side without cropping Approach used in the Multigrain paper https://arxiv.org/pdf/1902.05509.pdf

static target_size(*w: int*, *h: int*, *size: int*, *largest: bool = False*)

class vissl.utils.instance_retrieval_utils.data_util.**WhiteningTrainingImageDataset**(*base_dir: str*, *image_list_fil str*, *num_samp int* = *0*)

Bases: object

A set of training images for whitening

get_num_images()

get_filename(*i: int*)

class vissl.utils.instance_retrieval_utils.data_util.**InstreDataset**(*dataset_path: str*, *num_samples: int = 0*)

Bases: object

A dataset class that reads and parses the Instre Dataset so it's ready to be used in the code for retrieval evaluations

get_num_images()
Number of images in the dataset

get_num_query_images()
Number of query images in the dataset

get_filename(*i: int*)
Return the image filepath for the db image

get_query_filename(*i: int*)
Reutrn the image filepath for the query image

get_query_roi(*i: int*)
INSTRE dataset has no notion of ROI so we return None.

eval_from_ranks(*ranks*)
Return the mean average precision value or the train and validation both provided the ranks (scores of the model).

score(*scores*, *temp_dir*, *verbose=True*)
For the input scores of the model, calculate the AP metric

class vissl.utils.instance_retrieval_utils.data_util.**RevisitedInstanceRetrievalDataset**(*datas str*, *dir_n str*)

Bases: object

A dataset class used for the Revisited Instance retrieval datasets: Revisited Oxford and Revisited Paris. The object reads and parses the datasets so it's ready to be used in the code for retrieval evaluations.

get_filename(*i: int*)
Return the image filepath for the db image

get_query_filename(*i: int*)
Reutrn the image filepath for the query image

get_num_images()
Number of images in the dataset

get_num_query_images()
Number of query images in the dataset

get_query_roi(*i: int*)
Get the ROI for the query image that we want to test retrieval

score(*sim*, *temp_dir: str*)
For the input similarity scores of the model, calculate the mean AP metric and mean Precision@k metrics.

class vissl.utils.instance_retrieval_utils.data_util.**InstanceRetrievalImageLoader**(*S*, *transforms*)

Bases: object

The custom loader for the Paris and Oxford Instance Retrieval datasets.

apply_img_transform(*im*)
Apply the pre-defined transforms on the image.

load_and_prepare_whitening_image(*fname*)
from the filename, load the whitening image and prepare it to be used by applying data transforms

load_and_prepare_instre_image(*fname*)
from the filename, load the db or query image and prepare it to be used by applying data transforms

load_and_prepare_image(*fname*, *roi=None*)
Read image, get aspect ratio, and resize such as the largest side equals S. If there is a roi, adapt the roi to the new size and crop. Do not rescale the image once again. ROI format is (xmin,ymin,xmax,ymax)

load_and_prepare_revisited_image(*img_path*, *roi=None*)
Load the image, crop the roi from the image if the roi is not None, apply the image transforms.

class vissl.utils.instance_retrieval_utils.data_util.**InstanceRetrievalDataset**(*path*, *eval_binary_path*, *num_samples=No*

Bases: object

A dataset class used for the Instance retrieval datasets: Oxford and Paris. The object reads and parses the datasets so it's ready to be used in the code for retrieval evaluations.

Credits: https://github.com/facebookresearch/deepcluster/blob/master/eval_retrieval.py # NOQA Adapted by: Priya Goyal (prigoyal@fb.com)

get_num_images()
Number of images in the dataset

get_num_query_images()
Number of query images in the dataset

load(*num_samples=None*)
Load the data ground truth and parse the data so it's ready to be used.

score(*sim*, *temp_dir*)
From the input similarity score, compute the mean average precision

score_rnk_partial(*i*, *idx*, *temp_dir*)
Compute the mean AP for a given single query

get_filename(*i*)
Return the image filepath for the db image

get_query_filename(*i*)
Reutrn the image filepath for the query image

get_query_roi(*i*)
Get the ROI for the query image that we want to test retrieval

61.9.2 vissl.utils.instance_retrieval_utils.evaluate module

vissl.utils.instance_retrieval_utils.evaluate.**score_ap_from_ranks_1**(*ranks*, *nres*)
Compute the average precision of one search.

Parameters

- **ranks** – ordered list of ranks of true positives
- **nres** – total number of positives in dataset

Returns *ap (float)* – the average precision following the Holidays and the INSTRE package

vissl.utils.instance_retrieval_utils.evaluate.**compute_ap**(*ranks*, *nres*)
Computes average precision for given ranked indexes.

Parameters

- **ranks** – zero-based ranks of positive images
- **nres** – number of positive images

Returns *ap (float)* – average precision

vissl.utils.instance_retrieval_utils.evaluate.**compute_map**(*ranks*, *gnd*, *kappas*)
Computes the mAP for a given set of returned results.

Credits: https://github.com/filipradenovic/revisitop/blob/master/python/evaluate.py

Usage:

map = compute_map (ranks, gnd) computes mean average precsion (map) only

map, aps, pr, prs = compute_map (ranks, gnd, kappas)

-> computes mean average precision (map), average precision (aps) for each query

-> computes mean precision at kappas (pr), precision at kappas (prs) for each query

Notes: 1) ranks starts from 0, ranks.shape = db_size X #queries 2) The junk results (e.g., the query itself) should be declared in the gnd

stuct array

3) If there are no positive images for some query, that query is excluded from the evaluation

61.9.3 vissl.utils.instance_retrieval_utils.pca module

class vissl.utils.instance_retrieval_utils.pca.**PCA**(*n_components*)
Bases: object

Fits and applies PCA whitening

fit(*X*)

to_cuda()

apply(*X*)

vissl.utils.instance_retrieval_utils.pca.**load_pca**(*pca_out_fname*)

vissl.utils.instance_retrieval_utils.pca.**train_and_save_pca**(*features*, *n_pca*, *pca_out_fname*)

61.9.4 vissl.utils.instance_retrieval_utils.rmac module

vissl.utils.instance_retrieval_utils.rmac.**normalize_L2**(*a*, *dim*)
L2 normalize the input tensor along the specified dimension

Parameters

- **a** (*torch.Tensor*) – the tensor to normalize
- **dim** (*int*) – along which dimension to L2 normalize

Returns *a (torch.Tensor)* – L2 normalized tensor

vissl.utils.instance_retrieval_utils.rmac.**get_rmac_region_coordinates**(*H*, *W*, *L*)
Almost verbatim from Tolias et al Matlab implementation. Could be heavily pythonized, but really not worth it... Desired overlap of neighboring regions

vissl.utils.instance_retrieval_utils.rmac.**get_rmac_descriptors**(*features*, *rmac_levels*, *pca=None*)
RMAC descriptors. Coordinates are retrieved following Tolias et al. L2 normalize the descriptors and optionally apply PCA on the descriptors if specified by the user. After PCA, aggregate the descriptors (sum) and normalize the aggregated descriptor and return.

61.9.5 vissl.utils.svm_utils.evaluate module

vissl.utils.svm_utils.evaluate.**calculate_ap**(*rec*, *prec*)
Computes the AP under the precision recall curve.

vissl.utils.svm_utils.evaluate.**get_precision_recall**(*targets*, *scores*, *weights=None*)
[P, R, score, ap] = get_precision_recall(targets, scores, weights)

Parameters

- **targets** – number of occurrences of this class in the ith image
- **scores** – score for this image
- **weights** – 0 or 1 whether where 0 means we should ignore the sample

Returns *P, R* – precision and recall score: score which corresponds to the particular precision and recall ap: average precision

61.9.6 vissl.utils.svm_utils.svm_trainer module

class vissl.utils.svm_utils.svm_trainer.**SVMTrainer**(*config*, *layer*, *output_dir*)
Bases: object

SVM trainer that takes care of training (using k-fold cross validation), and evaluating the SVMs

load_input_data(*data_file*, *targets_file*)
Given the input data (features) and targets (labels) files, load the features of shape N x D and labels of shape (N,)

get_best_cost_value()
During the SVM training, we write the cross vaildation AP value for training at each class and cost value combination. We load the AP values and for each class, determine the cost value that gives the maximum AP. We return the chosen cost values for each class as a numpy matrix.

train_cls(*features*, *targets*, *cls_num*)
Train SVM on the input features and targets for a given class. The SVMs are trained for all costs values for the given class. We also save the cross-validation AP at each cost value for the given class.

train(*features*, *targets*)
Train SVMs on the given features and targets for all classes and all the costs values.

test(*features*, *targets*)
Test the trained SVM models on the test features and targets values. We use the cost per class that gives the maximum cross validation AP on the training and load the correspond trained SVM model for the cost value and the class.

Log the test ap to stdout and also save the AP in a file.

61.9.7 vissl.utils.svm_utils.svm_low_shot_trainer module

class vissl.utils.svm_utils.svm_low_shot_trainer.**SVMLowShotTrainer**(*config*, *layer*, *output_dir*)
Bases: *vissl.utils.svm_utils.svm_trainer.SVMTrainer*

Train the SVM for the low-shot image classification tasks. Currently, datasets like VOC07 and Places205 are supported.

The trained inherits from the SVMTrainer class and takes care of training SVM, evaluating, and aggregate the metrics.

train(*features*, *targets*, *sample_num*, *low_shot_kvalue*)
Train SVM on the input features and targets for a given low-shot k-value and the independent low-shot sample number.

We save the trained SVM model for each combination: cost value, class number, sample number, k-value

test(*features*, *targets*, *sample_num*, *low_shot_kvalue*)

Test the SVM for the input test features and targets for the given: low-shot k-value, sample number

We compute the meanAP across all classes for a given cost value. We get the output matrix of shape (1, #costs) for the given sample_num and k-value and save the matrix. We use this information to aggregate later.

aggregate_stats(*k_values*, *sample_inds*)
Aggregate the test AP across all k-values and independent samples.

For each low-shot k-value, we obtain the mean, max, min, std AP value. Steps:

1. For each k-value, get the min/max/mean/std value across all the independent samples. This results in matrices [#k-values x #classes]
2. Then we aggregate stats across the classes. For the mean stats in step 1, for each k-value, we get the class which has maximum mean.

61.9.8 vissl.utils.activation_checkpointing module

This module centralizes all activation checkpointing related code. It is a work-in-progress as we evolve the APIs and eventually put this in fairscale so that multiple projects can potentially share it.

vissl.utils.activation_checkpointing.**manual_gradient_reduction**(*model: torch.nn.modules.module.Module*, *config_flag: bool*) → bool

Return if we should use manual gradient reduction or not.

We should use manual DDP if config says so and model is wrapped by DDP.

vissl.utils.activation_checkpointing.**manual_sync_params**(*model: torch.nn.parallel.distributed.DistributedDataParall*) → None

Manually sync params and buffers for DDP.

vissl.utils.activation_checkpointing.**manual_gradient_all_reduce**(*model: torch.nn.parallel.distributed.Distributed*) → None

Gradient reduction function used after backward is done.

vissl.utils.activation_checkpointing.**layer_splittable_before**(*m: torch.nn.modules.module.Module*) → bool

Return if this module can be split in front of it for checkpointing. We don't split the relu module.

vissl.utils.activation_checkpointing.**checkpoint_trunk**(*feature_blocks: Dict[str, torch.nn.modules.module.Module]*, *unique_out_feat_keys: List[str]*, *checkpointing_splits: int*) → Dict[str, torch.nn.modules.module.Module]

Checkpoint a list of blocks and return back the split version.

61.9.9 vissl.utils.checkpoint module

vissl.utils.checkpoint.**is_training_finished**(*cfg:* vissl.utils.hydra_config.AttrDict, *checkpoint_folder: str*)

Given the checkpoint folder, we check that there's not already a final checkpoint If the final checkpoint exists but the user wants to override the final checkpoint then we mark training as not finished.

Parameters

- **cfg** (AttrDict) – input config file specified by user and parsed by vissl
- **checkpoint_folder** (*str*) – the directory where the checkpoints exist

Returns boolean whether training is finished or not.

vissl.utils.checkpoint.**get_checkpoint_folder**(*config:* vissl.utils.hydra_config.AttrDict)

Check, create and return the checkpoint folder. User can specify their own checkpoint directory otherwise the default "." is used.

Optionally, for training that involves more than 1 machine, we allow to append the distributed run id which helps to uniquely identify the training. This is completely optional and user can se APPEND_DISTR_RUN_ID=true for this.

vissl.utils.checkpoint.**is_checkpoint_phase**(*mode_num: int*, *mode_frequency: int*, *train_phase_idx: int*, *num_epochs: int*, *mode: str*)

Determines if a checkpoint should be saved on current epoch. If epoch=1, then we check whether to save at current iteration or not.

Parameters

- **mode** (*str*) – what model we are checkpointing models at - every few iterations or at the end of every phase/epoch. The mode is encoded in the checkpoint filename.
- **mode_num** (*int*) – what is the current iteration or epoch number that we are trying to checkpoint at.
- **mode_frequency** (*int*) – checkpoint frequency - every N iterations or every N epochs/phase
- **train_phase_idx** (*int*) – the current training phase we are in. Starts from 0
- **num_epochs** (*int*) – total number of epochs in training

Returns *checkpointing_phase (bool)* – whether the model should be checkpointed or not

vissl.utils.checkpoint.**has_checkpoint**(*checkpoint_folder: str*, *skip_final: bool = False*)

Check whether there are any checkpoints at all in the checkpoint folder.

Parameters

- **checkpoint_folder** (*str*) – path to the checkpoint folder
- **skip_final** (*bool*) – if the checkpoint with *model_final_* prefix exist, whether to skip it and train.

Returns *checkpoint_exists (bool)* – whether checkpoint exists or not

vissl.utils.checkpoint.**has_final_checkpoint**(*checkpoint_folder: str*, *final_checkpoint_pattern: str = 'model_final'*)

Check whether the final checkpoint exists in the checkpoint folder. The final checkpoint is recognized by the prefix "**model_final_**" in VISSL.

Parameters

- **checkpoint_folder** (*str*) – path to the checkpoint folder.
- **final_checkpoint_pattern** (*str*) – what prefix is used to save the final checkpoint.

Returns *has_final_checkpoint* – whether the final checkpoint exists or not

vissl.utils.checkpoint.**get_checkpoint_resume_files**(*checkpoint_folder: str*, *config:* vissl.utils.hydra_config.AttrDict, *skip_final: bool = False*, *latest_checkpoint_resume_num: int = 1*)

Get the checkpoint file from which the model should be resumed. We look at all the checkpoints in the checkpoint_folder and if the final model checkpoint exists (starts with *model_final_*) and not overriding it, then return the final checkpoint. Otherwise find the latest checkpoint.

Parameters

- **checkpoint_folder** (*str*) – path to the checkpoint folder.
- **config** (AttrDict) – root config
- **skip_final** (*bool*) – whether the final model checkpoint should be skipped or not
- **latest_checkpoint_resume_num** (*int*) – what Nth latest checkpoint to resume from. Sometimes the latest checkpoints could be corrupt so this option helps to resume from instead a few checkpoints before the last checkpoint.

vissl.utils.checkpoint.**get_resume_checkpoint**(*cfg: vissl.utils.hydra_config.AttrDict, checkpoint_folder: str*)

Return the checkpoint from which to resume traning. If no checkpoint found, return None. Resuming training is optional and user can set AUTO_RESUME=false to not resume the training.

If we want to overwrite the existing final checkpoint, we ignore the final checkpoint and return the previous checkpoints if exist.

vissl.utils.checkpoint.**print_state_dict_shapes**(*state_dict: Dict[str, Any]*)

For the given model state dictionary, print the name and shape of each parameter tensor in the model state. Helps debugging.

Parameters **state_dict** (*Dict[str, Any]*) – model state dictionary

vissl.utils.checkpoint.**print_loaded_dict_info**(*model_state_dict: Dict[str, Any], state_dict: Dict[str, Any], skip_layers: List[str], model_config: vissl.utils.hydra_config.AttrDict*)

Print what layers were loaded, what layers were ignored/skipped/not found when initializing a model from a specified model params file.

vissl.utils.checkpoint.**replace_module_prefix**(*state_dict: Dict[str, Any], prefix: str, replace_with: str = ''*)

Remove prefixes in a state_dict needed when loading models that are not VISSL trained models.

Specify the prefix in the keys that should be removed.

vissl.utils.checkpoint.**append_module_prefix**(*state_dict: Dict[str, Any], prefix: str*)

Append prefixes in a state_dict needed when loading models that are not VISSL trained models.

In order to load the model (if not trained with VISSL) with VISSL, there are 2 scenarios:

1. If you are interested in evaluating the model features and freeze the trunk. Set APPEND_PREFIX="trunk.base_model." This assumes that your model is compatible with the VISSL trunks. The VISSL trunks start with "_feature_blocks." prefix. If your model doesn't have these prefix you can append them. For example: For TorchVision ResNet trunk, set APPEND_PREFIX="trunk.base_model._feature_blocks."
2. where you want to load the model simply and finetune the full model. Set APPEND_PREFIX="trunk." This assumes that your model is compatible with the VISSL trunks. The VISSL trunks start with "_feature_blocks." prefix. If your model doesn't have these prefix you can append them. For TorchVision ResNet trunk, set APPEND_PREFIX="trunk._feature_blocks."

NOTE: the prefix is appended to all the layers in the model

vissl.utils.checkpoint.**check_model_compatibilty**(*config: vissl.utils.hydra_config.AttrDict, state_dict: Dict[str, Any]*)

Given a VISSL model and state_dict, check if the state_dict can be loaded to VISSL model (trunk + head) based on the trunk and head prefix that is expected. If not compatible, we raise exception.

Prefix checked for head: *heads.* Prefix checked for trunk: *trunk._feature_blocks.* or *trunk.base_model._feature_blocks.*

depending on the workflow type (training | evaluation).

Parameters

- **config** (AttrDict) – root config
- **state_dict** (*Dict[str, Any]*) – state dict that should be checked for compatibility

vissl.utils.checkpoint.**get_checkpoint_model_state_dict**(*config:* vissl.utils.hydra_config.AttrDict, *state_dict: Dict[str, Any]*)

Given a specified pre-trained VISSL model (composed of head and trunk), we get the state_dict that can be loaded by appending prefixes to model and trunk.

Parameters

- **config** (AttrDict) – full config file
- **state_dict** (*Dict*) – raw state_dict loaded from the checkpoint or weights file

Returns

state_dict (Dict) –

vissl state_dict with layer names matching compatible with vissl model. Hence this state_dict can be loaded directly.

vissl.utils.checkpoint.**init_model_from_weights**(*config:* vissl.utils.hydra_config.AttrDict, *model*, *state_dict: Dict[str, Any]*, *state_dict_key_name: str*, *skip_layers: List[str]*, *replace_prefix=None*, *append_prefix=None*)

Initialize the model from any given params file. This is particularly useful during the feature evaluation process or when we want to evaluate a model on a range of tasks.

Parameters

- **config** (AttrDict) – config file
- **model** (*object*) – instance of base_ssl_model
- **state_dict** (*Dict*) – torch.load() of user provided params file path.
- **state_dict_key_name** (*string*) – key name containing the model state dict
- **skip_layers** (*List(string)*) – layer names with this key are not copied
- **replace_prefix** (*string*) – remove these prefixes from the layer names (executed first)
- **append_prefix** (*string*) – append the prefix to the layer names (executed after replace_prefix)

Returns *model (object)* – the model initialized from the weights file

61.9.10 vissl.utils.collect_env module

vissl.utils.collect_env.**collect_env_info**()
Collect information about user system including cuda, torch, gpus, vissl and its dependencies. Users are strongly recommended to run this script to collect information about information if they needed debugging help.

61.9.11 vissl.utils.env module

vissl.utils.env.**set_env_vars**(*local_rank: int*, *node_id: int*, *cfg: vissl.utils.hydra_config.AttrDict*)
Set some environment variables like total number of gpus used in training, distributed rank and local rank of the current gpu, whether to print the nccl debugging info and tuning nccl settings.

vissl.utils.env.**print_system_env_info**(*current_env*)
Print information about user system environment where VISSL is running.

vissl.utils.env.**get_machine_local_and_dist_rank**()
Get the distributed and local rank of the current gpu.

61.9.12 vissl.utils.hydra_config module

class vissl.utils.hydra_config.**AttrDict**(*dictionary*)
Bases: dict

Dictionary subclass whose entries can be accessed like attributes (as well as normally). Credits: https://aiida.readthedocs.io/projects/aiida-core/en/latest/_modules/aiida/common/extendeddicts.html#AttributeDict # noqa

__init__(*dictionary*)
Recursively turn the *dict* and all its nested dictionaries into *AttrDict* instance.

__getattr__(*key*)
Read a key as an attribute.

Raises **AttributeError** – if the attribute does not correspond to an existing key.

__setattr__(*key*, *value*)
Set a key as an attribute.

__delattr__(*key*)
Delete a key as an attribute.

Raises **AttributeError** – if the attribute does not correspond to an existing key.

__getstate__()
Needed for pickling this class.

__setstate__(*dictionary*)
Needed for pickling this class.

__deepcopy__(*memo=None*)
Deep copy.

vissl.utils.hydra_config.**convert_to_attrdict**(*cfg: omegaconf.dictconfig.DictConfig*, *cmd-line_args: List[Any] = None*)
Given the user input Hydra Config, and some command line input options to override the config file: 1. merge and override the command line options in the config 2. Convert the Hydra OmegaConf to AttrDict structure to make it easy

to access the keys in the config file

3. Also check the config version used is compatible and supported in vissl. In future, we would want to support upgrading the old config versions if we make changes to the VISSL default config structure (deleting, renaming keys)
4. We infer values of some parameters in the config file using the other parameter values.

vissl.utils.hydra_config.**is_hydra_available**()
Check if Hydra is available. Simply python import to test.

vissl.utils.hydra_config.**print_cfg**(*cfg*)
Supports printing both Hydra DictConfig and also the AttrDict config

vissl.utils.hydra_config.**resolve_linear_schedule**(*cfg*, *param_schedulers*)
For the given composite schedulers, for each linear schedule, if the training is 1 node only, the https://arxiv.org/abs/1706.02677 linear warmup rule has to be checked if the rule is applicable and necessary.

We set the end_value = scaled_lr (assuming it's a linear warmup). In case only 1 machine is used in training, the start_lr = scaled_lr and then the linear warmup is not needed.

vissl.utils.hydra_config.**get_scaled_lr_scheduler**(*cfg*, *param_schedulers*, *scaled_lr*)
Scale learning rate value for different Learning rate types. See assert_learning_rate() for how the scaled LR is calculated.

Values changed for learning rate schedules: 1. cosine:

end_value = scaled_lr * (end_value / start_value) start_value = scaled_lr and

2. **multistep:** gamma = values[1] / values[0] values = [scaled_lr * pow(gamma, idx) for idx in range(len(values))]
3. **step_with_fixed_gamma** base_value = scaled_lr
4. linear: end_value = scaled_lr
5. inverse_sqrt: start_value = scaled_lr
6. constant: value = scaled_lr
7. **composite:** recursively call to scale each composition. If the composition consists of a linear schedule, we assume that a linear warmup is applied. If the linear warmup is applied, it's possible the warmup is not necessary if the global batch_size is smaller than the base_lr_batch_size and in that case, we remove the linear warmup from the schedule.

vissl.utils.hydra_config.**assert_learning_rate**(*cfg*)
1) Assert the Learning rate here. LR is scaled as per https://arxiv.org/abs/1706.02677. to turn this automatic scaling off, set config.OPTIMIZER.param_schedulers.lr.auto_lr_scaling.auto_scale=false

scaled_lr is calculated:

given base_lr_batch_size = batch size for which the base learning rate is specified, base_value = base learning rate value that will be scaled, The current batch size is used to determine how to scale the base learning rate value.

scaled_lr = ((batchsize_per_gpu * world_size) * base_value) / base_lr_batch_size

We perform this auto-scaling for head learning rate as well if user wants to use a different learning rate for the head

2) infer the model head params weight decay: if the head should use a different weight decay value than the trunk. If using different weight decay value for the head, set here. otherwise, the same value as trunk will be automatically used.

vissl.utils.hydra_config.**assert_losses**(*cfg*)
Infer settings for various self-supervised losses. Takes care of setting various loss parameters correctly like world size, batch size per gpu, effective global batch size, collator etc. Each loss has additional set of parameters that can be inferred to ensure smooth training in case user forgets to adjust all the parameters.

vissl.utils.hydra_config.**assert_hydra_conf**(*cfg*)
Infer values of few parameters in the config file using the value of other config parameters 1. Inferring losses 2. Auto scale learning rate if user has specified auto scaling to be True. 3. Infer meter names (model layer name being evaluated) since we support list meters

that have multiple output and same target. This is very common in self-supervised learning where we want to evaluate metric for several layers of the models. VISSL supports running evaluation for multiple model layers in a single training run.

4. Support multi-gpu DDP eval model by attaching a dummy parameter. This is particularly helpful for the multi-gpu feature extraction especially when the dataset is large for which features are being extracted.
5. Infer what kind of labels are being used. If user has specified a labels source, we set LABEL_TYPE to "standard" (also vissl default), otherwise if no label is specified, we set the LABEL_TYPE to "sample_index".

61.9.13 vissl.utils.io module

vissl.utils.io.**cache_url**(*url: str*, *cache_dir: str*) → str
This implementation downloads the remote resource and caches it locally. The resource will only be downloaded if not previously requested.

vissl.utils.io.**create_file_symlink**(*file1*, *file2*)
Simply create the symlinks for a given file1 to file2. Useful during model checkpointing to symlinks to the latest successful checkpoint.

vissl.utils.io.**save_file**(*data*, *filename*)
Common i/o utility to handle saving data to various file formats. Supported:

.pkl, .pickle, .npy, .json

vissl.utils.io.**load_file**(*filename*, *mmap_mode=None*)
Common i/o utility to handle loading data from various file formats. Supported:

.pkl, .pickle, .npy, .json

For the npy files, we support reading the files in mmap_mode. If the mmap_mode of reading is not successful, we load data without the mmap_mode.

vissl.utils.io.**makedir**(*dir_path*)
Create the directory if it does not exist.

vissl.utils.io.**is_url**(*input_url*)
Check if an input string is a url. look for http(s):// and ignoring the case

vissl.utils.io.**cleanup_dir**(*dir*)
Utility for deleting a directory. Useful for cleaning the storage space that contains various training artifacts like checkpoints, data etc.

vissl.utils.io.**get_file_size**(*filename*)
Given a file, get the size of file in MB

vissl.utils.io.**copy_file**(*input_file*, *destination_dir*, *tmp_destination_dir*)
Copy a given input_file from source to the destination directory.

Steps: 1. We use PathManager to extract the data to local path. 2. we simply move the files from the PathManager cached local directory

to the user specified destination directory. We use rsync. How destination dir is chosen:

a) If user is using slurm, we set destination_dir = slurm_dir (see get_slurm_dir)

b) If the local path used by PathManafer is same as the input_file path, and the destination directory is not specified, we set destination_dir = tmp_destination_dir

Returns *output_file (str)* – the new path of the file destination_dir (str): the destination dir that was actually used

vissl.utils.io.**copy_dir**(*input_dir*, *destination_dir*, *num_threads*)

Copy contents of one directory to the specified destination directory using the number of threads to speed up the copy. When the data is copied successfully, we create a copy_complete file in the destination_dir folder to mark the completion. If the destination_dir folder already exists and has the copy_complete file, we don't copy the file.

useful for copying datasets like ImageNet to speed up dataloader. Using 20 threads for imagenet takes about 20 minutes to copy.

Returns *destination_dir (str)* – directory where the contents were copied

vissl.utils.io.**copy_data**(*input_file*, *destination_dir*, *num_threads*, *tmp_destination_dir*)

Copy data from one source to the other using num_threads. The data to copy can be a single file or a directory. We check what type of data and call the relevant functions.

Returns *output_file (str)* – the new path of the data (could be file or dir) destination_dir (str): the destination dir that was actually used

vissl.utils.io.**copy_data_to_local**(*input_files*, *destination_dir*, *num_threads=40*, *tmp_destination_dir=None*)

Iteratively copy the list of data to a destination directory. Each data to copy could be a single file or a directory.

Returns

output_file (str) –

the new path of the file. If there were no files to copy, simply return the input_files

destination_dir (str): the destination dir that was actually used

61.9.14 vissl.utils.logger module

vissl.utils.logger.**setup_logging**(*name*, *output_dir=None*, *rank=0*)

Setup various logging streams: stdout and file handlers.

For file handlers, we only setup for the master gpu.

vissl.utils.logger.**shutdown_logging**()

After training is done, we ensure to shut down all the logger streams.

vissl.utils.logger.**log_gpu_stats**()

Log nvidia-smi snapshot. Useful to capture the configuration of gpus.

vissl.utils.logger.**print_gpu_memory_usage**()

Parse the nvidia-smi output and extract the memory used stats. Not recommended to use.

61.9.15 vissl.utils.misc module

vissl.utils.misc.**is_apex_available**()
Check if apex is available with simple python imports.

vissl.utils.misc.**is_faiss_available**()
Check if faiss is available with simple python imports. To install faiss, simply do:

If using PIP env: *pip install faiss-gpu* If using conda env: *conda install faiss-gpu -c pytorch*

vissl.utils.misc.**is_opencv_available**()
Check if opencv is available with simple python imports. To install opencv, simply do: *pip install opencv-python* regardless of whether using conda or pip environment.

vissl.utils.misc.**find_free_tcp_port**()
Find the free port that can be used for Rendezvous on the local machine. We use this for 1 machine training where the port is automatically detected.

vissl.utils.misc.**get_dist_run_id**(*cfg*, *num_nodes*)
For multi-gpu training with PyTorch, we have to specify how the gpus are going to rendezvous. This requires specifying the communication method: file, tcp and the unique rendezvous run_id that is specific to 1 run.

We recommend:

1) for 1-node: use init_method=tcp and run_id=auto

2) for multi-node, use init_method=tcp and specify run_id={master_node}:{port}

vissl.utils.misc.**setup_multiprocessing_method**(*method_name: str*)
PyTorch supports several multiprocessing options: forkserver | spawn | fork

We recommend and use forkserver as the default method in VISSL.

vissl.utils.misc.**set_seeds**(*cfg*, *node_id=0*)
Set the python random, numpy and torch seed for each gpu. Also set the CUDA seeds if the CUDA is available. This ensures deterministic nature of the training.

vissl.utils.misc.**get_indices_sparse**(*data*)
Is faster than np.argwhere. Used in loss functions like swav loss, etc

vissl.utils.misc.**merge_features**(*output_dir*, *split*, *layer*, *cfg*)
For multi-gpu feature extraction, each gpu saves features corresponding to its share of the data. We can merge the features across all gpus to get the features for the full data.

The features are saved along with the data indexes and label. The data indexes can be used to sort the data and ensure the uniqueness.

We organize the features, targets corresponding to the data index of each feature, ensure the uniqueness and return.

Parameters

- **output_dir** (*str*) – input path where the features are dumped
- **split** (*str*) – whether the features are train or test data features
- **layer** (*str*) – the features correspond to what layer of the model
- **cfg** (AttrDict) – the input configuration specified by user

Returns *output (Dict)* – contains features, targets, inds as the keys

vissl.utils.misc.**get_json_data_catalog_file**()
Searches for the dataset_catalog.json file that contains information about the dataset paths if set by user.

vissl.utils.misc.**concat_all_gather**(*tensor*)
Performs all_gather operation on the provided tensors. * **Warning** *: torch.distributed.all_gather has no gradient.

61.9.16 vissl.utils.perf_stats module

class vissl.utils.perf_stats.**PerfTimer**(*timer_name: str*, *perf_stats: Optional[PerfStats]*)
Bases: object

Very simple timing wrapper, with context manager wrapping. Typical usage:

with PerfTimer('forward_pass', perf_stats): model.forward(data)

... with PerfTimer('backward_pass', perf_stats):

model.backward(loss)

... print(perf_stats.report_str())

Note that timer stats accumulate by name, so you can as if resume them by re-using the name.

You can also use it without context manager, i.e. via start() / stop() directly.

If supplied PerfStats is constructed with use_cuda_events=True (which is default), then Cuda events will be added to correctly track time of async execution of Cuda kernels:

with PerfTimer('foobar', perf_stats): some_cpu_work() schedule_some_cuda_work()

In example above, the "Host" column will capture elapsed time from the perspective of the Python process, and "CudaEvent" column will capture elapsed time between scheduling of Cuda work (within the PerfTimer scope) and completion of this work, some of which might happen outside the PerfTimer scope.

If perf_stats is None, using PerfTimer does nothing.

start()
Start the recording if the perfTimer should not be skipped or if the recording is not already in progress. If using cuda, we record time of cuda events as well.

stop()
Stop the recording and update the recording interaval, total time elapsed from the beginning of perfTimer recording. If using CUDA, we measure time for cuda events and append to cuda interval.

record()
Update the timer. We should only do this if the timer is Not skipped and also if the timer has already been stopped.

class vissl.utils.perf_stats.**PerfMetric**
Bases: object

Encapsulates numerical tracking of a single metric, with a *.update(value)* API. Under-the-hood this can additionally keep track of sums, (exp.) moving averages, sum of squares (e.g. for stdev), filtered values, etc.

EMA_FACTOR = 0.1

update(*value: float*)

get_avg()
Get the mean value of the metrics recorded.

class vissl.utils.perf_stats.**PerfStats**(*use_cuda_events=True*)
Bases: object

Accumulate stats (from timers) over many iterations

MAX_PENDING_TIMERS = 1000

update_with_timer(*timer:* vissl.utils.perf_stats.PerfTimer)

report_str()
Fancy column-aligned human-readable report. If using Cuda events, calling this invokes cuda.synchronize(), which is needed to capture pending Cuda work in the report.

use_cuda_events()

61.9.17 vissl.utils.slurm module

vissl.utils.slurm.**get_node_id**(*node_id: int*)
If using SLURM, we get environment variables like SLURMD_NODENAME, SLURM_NODEID to get information about the current node. Useful to set the node_id automatically.

vissl.utils.slurm.**get_slurm_dir**(*input_dir: str*)
If using SLURM, we use the environment variable "SLURM_JOBID" to uniquely identify the current training and append the id to the input directory. This could be used to store any training artifacts specific to this training run.

61.9.18 vissl.utils.tensorboard module

This script contains some helpful functions to handle tensorboard setup.

vissl.utils.tensorboard.**is_tensorboard_available**()
Check whether tensorboard is available or not.

Returns

tb_available (bool) –

based on tensorboard imports, returns whether tensboarboard is available or not.

vissl.utils.tensorboard.**get_tensorboard_dir**(*cfg*)
Get the output directory where the tensorboard events will be written.

Parameters **cfg** (AttrDict) – User specified config file containing the settings for the tensorboard as well like log directory, logging frequency etc

Returns *tensorboard_dir (str)* – output directory path

vissl.utils.tensorboard.**get_tensorboard_hook**(*cfg*)
Construct the Tensorboard hook for visualization from the specified config

Parameters **cfg** (AttrDict) – User specified config file containing the settings for the tensorboard as well like log directory, logging frequency etc

Returns *SSLTensorboardHook (function)* – the tensorboard hook constructed

PYTHON MODULE INDEX

V

INDEX

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W