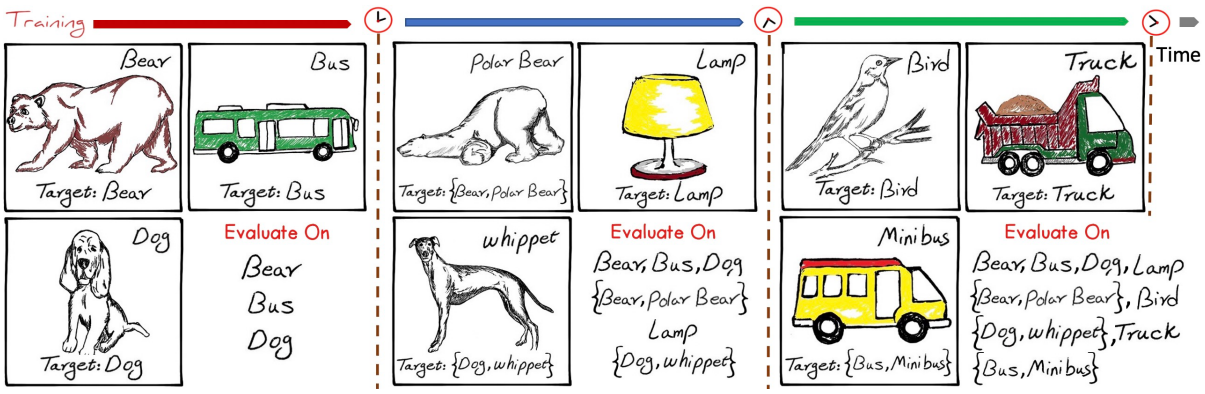
IIRC

Release 1.0.0

Mohamed Abdelsalam, Mojtaba Faramarzi, Shagun Sodhani, Sara

Jan 20, 2021

CONTENTS:



iirc is a package for adapting the different datasets (currently supports *CIFAR-100* and *ImageNet*) to the *iirc* setup and the *class incremental learning* setup, and loading them in a standardized manner.

lifelong_methods is a package that standardizes the different stages any lifelong learning method passes by, hence it provides a faster way for implementing new ideas and embedding them in the same training code as other baselines, it provides as well the implementation of some of these baselines.

Project Homepage | Project Paper | Source Code | PyPI Package

CHAPTER

ONE

INSTALLATION:

you can install the **iirc** package using the following command

```
pip install iirc
```

To use it with PyTorch, you will need as well to install *PyTorch* (1.5.0) and *torchvision* (0.6.0)

CHAPTER

TWO

GUIDE:

2.1 IIRC Package Tutorial Using PyTorch

[1]:
```
import numpy as np
from PIL import Image

from iirc.lifelong_dataset.torch_dataset import Dataset
from iirc.definitions import IIRC_SETUP, CIL_SETUP, NO_LABEL_PLACEHOLDER
```

2.1.1 Using CIL setup (Class Incremental Learning)

Let's create a mock dataset of 120 samples, with each belonging to one of the four classes **A** - **B** - **C** - **D**

[2]:
```
n = 120
n_per_y = 30
x = np.random.rand(n,32,32,3)
y = ["A"] * n_per_y + ["B"] * n_per_y + ["C"] * n_per_y + ["D"] * n_per_y
```

Now this dataset should be converted to the format used by the lifelong_datasets, where images should be a pillow images (or strings representing the images path in case of large datasets, such as ImageNet), and the dataset should be arranged as a list of tuples of the form (image, (label,))

[3]:
```
x = np.uint8(x*255)
mock_dataset = [(Image.fromarray(x[i], 'RGB'), (y[i],)) for i in range(n)]
```

Now let's create a tasks schedule, where the first task introduces the labels **A** and **B**, and the second task introduces **C** and **D** (tasks don't need to be of equal size)

[4]:
```
tasks = [["A", "B"], ["C", "D"]]
```

We also need a transformations function that takes the image and converts it to a tensor, as well as normalize the image, apply augmentations, etc.

There are two such functions that can be provided: *essential_transforms_fn* and *augmentation_transforms_fn*

If *augmentation_transforms_fn* is provided, it will always be applied except if the *Dataset* is told not to apply augmentations in a specific context (we will see later how)

Otherwise, *essential_transforms_fn* will be applied

So for example in a test set where augmentations are not needed, *augmentation_transforms_fn* shouldn't be provided in the *Dataset* initialization

Hence, *essential_transforms_fn* should include any essential transformations that should be applied to the PIL image (such as convert to tensor), while *augmentation_transforms_fn* should also include the essential transformations, in addition to any augmentations that need to be applied (such as random horizontal flipping, etc)

[5]:
```
import torchvision.transforms as transforms

essential_transforms_fn = transforms.ToTensor()
augmentation_transforms_fn = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()
])
```

Now we are ready to initialize the incremental dataset

[6]:
```
cil_dataset = Dataset(dataset=mock_dataset, tasks=tasks, setup=CIL_SETUP,
                              essential_transforms_fn=essential_transforms_fn,
                              augmentation_transforms_fn=augmentation_transforms_fn)
```

Then we can make use of *cil_dataset* by choosing the task we wish to train on using the *choose_task(task_id)* function, and creating a dataloader out of it

[7]:
```
cil_dataset.choose_task(0)
```

If we print the length of the dataset, we will only get the length of the samples that belong to the current task. The same goes for fetching a sample, as we only have access to the samples of the current task. This means that indexing the *cil_dataset* is relative to the current task, so the 0th sample for example will be different when we choose another task

[8]:
```
print(len(cil_dataset))
```
```
60
```

[9]:
```
cil_dataset[0]
```
[9]:
```
(tensor([[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
          [0.8431, 0.6902, 0.9294,  ..., 0.0000, 0.0000, 0.0000],
          [0.5922, 0.0510, 0.6314,  ..., 0.0000, 0.0000, 0.0000],
          ...,
          [0.3804, 0.2863, 0.5451,  ..., 0.0000, 0.0000, 0.0000],
          [0.2235, 0.8431, 0.8118,  ..., 0.0000, 0.0000, 0.0000],
          [0.8275, 0.8667, 0.6471,  ..., 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
          [0.9294, 0.3882, 0.2745,  ..., 0.0000, 0.0000, 0.0000],
          [0.4392, 0.7098, 0.7922,  ..., 0.0000, 0.0000, 0.0000],
          ...,
          [0.7176, 0.5725, 0.0118,  ..., 0.0000, 0.0000, 0.0000],
          [0.9137, 0.6471, 0.9882,  ..., 0.0000, 0.0000, 0.0000],
          [0.3804, 0.4667, 0.3216,  ..., 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
          [0.2353, 0.2824, 0.5882,  ..., 0.0000, 0.0000, 0.0000],
          [0.5765, 0.5490, 0.5686,  ..., 0.0000, 0.0000, 0.0000],
          ...,
          [0.5020, 0.0941, 0.0745,  ..., 0.0000, 0.0000, 0.0000],
          [0.3412, 0.8392, 0.2745,  ..., 0.0000, 0.0000, 0.0000],
          [0.9922, 0.0627, 0.2314,  ..., 0.0000, 0.0000, 0.0000]]]),
```

(continues on next page)

(continued from previous page)

```
 'A',
 'None')
```

As we have seen the format returned when we fetch a sample from the dataset is a tuple of length 3 consisting of (image tensor, image label, NO_LABEL_PLACEHOLDER), NO_LABEL_PLACEHOLDER is set to the string "None" and should be ignored in the class incremental setup.

We can also access what classes have we seen thus far (they don't reset if a previous task is re-chosen, also they are unordered):

[10]:
```
cil_dataset.reset()
cil_dataset.choose_task(0)
print(f"Task {cil_dataset.cur_task_id}, current task: {cil_dataset.cur_task}, dataset␣
↪length: {len(cil_dataset)}, classes seen: {cil_dataset.seen_classes}")
cil_dataset.choose_task(1)
print(f"Task {cil_dataset.cur_task_id}, current task: {cil_dataset.cur_task}, dataset␣
↪length: {len(cil_dataset)}, classes seen: {cil_dataset.seen_classes}")
```

```
Task 0, current task: ['A', 'B'], dataset length: 60, classes seen: ['A', 'B']
Task 1, current task: ['C', 'D'], dataset length: 60, classes seen: ['C', 'A', 'B', 'D
↪']
```

If we need to load the data of all the tasks up to a specific task (including it), this can be done using the *load_tasks_up_to(task_id)* function

[11]:
```
cil_dataset.load_tasks_up_to(1)
print(f"current task: {cil_dataset.cur_task}, dataset length: {len(cil_dataset)}")
```

```
current task: ['A', 'B', 'C', 'D'], dataset length: 120
```

If a sample needs to be accessed in it's original form (PIL form) without any transformations, for example to be added to the replay buffer, the *get_item(index)* method can be used:

[12]:
```
cil_dataset.get_item(0)
```

[12]:
```
(<PIL.Image.Image image mode=RGB size=32x32 at 0x22D7FF5FF10>, 'A', 'None')
```

and if we need to know the indices of the samples that belong to a specific class, this can be done using the *get_image_indices_by_cla(class name)* method, however, this can only be done if that class belongs to the current task. Moreover, these indices are relative to the current task, so whenever we change the task, they would point to totally different samples in the new task

[13]:
```
cil_dataset.get_image_indices_by_cla("A")
```

[13]:
```
array([29,  3, 23, 17, 10, 28,  2, 12, 22,  5, 19,  1, 25,  8,  4, 27,  7,
       16, 11, 21, 24, 15, 18,  9, 13, 26, 20, 14,  6,  0])
```

If *cil_dataset* uses data augmentations, but we needed to disable them in a specific part of the code, this context manager can be used:

[14]:
```
with cil_dataset.disable_augmentations():
    # any samples loaded here will have the essential transformations function␣
↪applied to them
    pass
```

Finally, to load this dataset in minibatches for training, the torch dataloader can be used with it, **but don't forget to reinstantiate the dataloader whenever the task changes**

[15]:
```
from torch.utils.data import DataLoader
cil_dataset.choose_task(0)
train_loader = DataLoader(cil_dataset, batch_size=2, shuffle=True)
next(iter(train_loader))
```

[15]:
```
[tensor([[[[0.0000, 0.0000, 0.0000,  ..., 0.9059, 0.3020, 0.3294],
           [0.0000, 0.0000, 0.0000,  ..., 0.6588, 0.7255, 0.1373],
           [0.0000, 0.0000, 0.0000,  ..., 0.5725, 0.1137, 0.7373],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]],

          [[0.0000, 0.0000, 0.0000,  ..., 0.4510, 0.6980, 0.2000],
           [0.0000, 0.0000, 0.0000,  ..., 0.6941, 0.6784, 0.2353],
           [0.0000, 0.0000, 0.0000,  ..., 0.8392, 0.0980, 0.1843],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]],

          [[0.0000, 0.0000, 0.0000,  ..., 0.9216, 0.9725, 0.4549],
           [0.0000, 0.0000, 0.0000,  ..., 0.1529, 0.8157, 0.2745],
           [0.0000, 0.0000, 0.0000,  ..., 0.8941, 0.1765, 0.1412],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],


         [[[0.0000, 0.0000, 0.0000,  ..., 0.3843, 0.1843, 0.8078],
           [0.0000, 0.0000, 0.0000,  ..., 0.0588, 0.1961, 0.5882],
           [0.0000, 0.0000, 0.0000,  ..., 0.5843, 0.0275, 0.1725],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]],

          [[0.0000, 0.0000, 0.0000,  ..., 0.5961, 0.4549, 0.3137],
           [0.0000, 0.0000, 0.0000,  ..., 0.7059, 0.2902, 0.2706],
           [0.0000, 0.0000, 0.0000,  ..., 0.5412, 0.3412, 0.7098],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]],

          [[0.0000, 0.0000, 0.0000,  ..., 0.5098, 0.0902, 0.6667],
           [0.0000, 0.0000, 0.0000,  ..., 0.1216, 0.2706, 0.7922],
           [0.0000, 0.0000, 0.0000,  ..., 0.5294, 0.1098, 0.5294],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]]]),
 ('B', 'A'),
 ('None', 'None')]
```

2.1.2 Using IIRC setup (Incremental Implicitly Refined Classification)

Now Imaging that samples belonging to class **A** have also another sublabel of either **Aa** or **Ab**, and the samples belonging to class **B** have another sublabel of either **Ba** or **Bb**

[16]:
```
y_iirc = []
for i, label in enumerate(y):
    if label == "A" and (i % 2) == 0:
        y_iirc.append(("A", "Aa"))
    elif label == "A":
        y_iirc.append(("A", "Ab"))
    elif label == "B" and (i % 2) == 0:
        y_iirc.append(("B", "Ba"))
    elif label == "B":
        y_iirc.append(("B", "Bb"))
    else:
        y_iirc.append((label,))
print(y_iirc)
```

```
[('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A',
↪'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'),
↪ ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A',
↪ 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab
↪'), ('A', 'Aa'), ('A', 'Ab'), ('A', 'Aa'), ('A', 'Ab'), ('B', 'Ba'), ('B', 'Bb'), (
↪'B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B',
↪'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'),
↪ ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B',
↪ 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb'), ('B', 'Ba'), ('B', 'Bb
↪'), ('B', 'Ba'), ('B', 'Bb'), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',
↪), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), (
↪'C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',), ('C',
↪), ('C',), ('C',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), (
↪'D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',
↪), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), ('D',), (
↪'D',)]
```

[17]:
```
mock_dataset_iirc = [(Image.fromarray(x[i], 'RGB'), y_iirc[i]) for i in range(n)]
```

[18]:
```
mock_dataset_iirc[0]
```

[18]:
```
(<PIL.Image.Image image mode=RGB size=32x32 at 0x22D7FF79B80>, ('A', 'Aa'))
```

let's redefine the tasks by incorporating the new subclasses

[19]:
```
tasks_iirc = [["A", "B", "C"], ["Aa", "Ba", "D"], ["Ab", "Bb"]]
```

All the functionality that we mentioned in the CIL setup is still applicable here, but there are some differences though.

The first difference is that in the training set, we don't necessarily need all the samples that belong to both labels **A** and **Aa** to be seen across the two tasks, what makes more sense is that some of them need to appear in the first task and some others need to appear in the second task, probably with some overlap

Hence we have the two arguments *superclass_data_pct* and *subclass_data_pct*: * *superclass_data_pct* controls the percentage of the samples that belong to **A** that will appear when **A** is introduced * *subclass_data_pct* controls the percentage of the samples that belong to **Aa** that will appear when **Aa** is introduced (same for other subclasses)

So in this example we have 15 samples that have the labels (**A**, **Aa**), and 15 samples that have the labels (**A**, **Ab**): * *superclass_data_pct* = 1.0, *subclass_data_pct* = 1.0: This means that all 30 samples with label **A** will appear in the first task, then all 15 samples with label **Aa** will appear in the second task, etc (100% overlab) * *superclass_data_pct*

= 0.5, *subclass_data_pct* = 0.5: 15 samples with label **A** will appear in the first task, then 8 samples with label **Aa** will appear in the second task, etc (no overlab) * *superclass_data_pct* = 0.6, *subclass_data_pct* = 0.8: 18 samples with label **A** will appear in the first task, then 12 samples with label **Aa** will appear in the second task, etc (40% overlab)

This procedure will only be done if the argument *test_mode* is set to *False*, as in the test set we don't care about this kind of data repetition but we care more about evaluating on all the available samples

[20]:
```
iirc_dataset_train = Dataset(dataset=mock_dataset_iirc, tasks=tasks_iirc, setup=IIRC_
↪SETUP, test_mode=False,
                                    essential_transforms_fn=essential_transforms_fn,
                                    augmentation_transforms_fn=augmentation_
↪transforms_fn,
                                    superclass_data_pct=0.6, subclass_data_pct=0.8)

iirc_dataset_test = Dataset(dataset=mock_dataset_iirc, tasks=tasks_iirc, setup=IIRC_
↪SETUP, test_mode=True,
                                    essential_transforms_fn=essential_transforms_fn,
                                    augmentation_transforms_fn=augmentation_
↪transforms_fn)

iirc_dataset_train.choose_task(0)
iirc_dataset_test.choose_task(0)

print(f"Length of task 0 training data: {len(iirc_dataset_train)}, number of samples␣
↪for class A: {len(iirc_dataset_train.get_image_indices_by_cla('A'))}")
print(f"Length of task 0 training data: {len(iirc_dataset_train)}, number of samples␣
↪for class A: {len(iirc_dataset_test.get_image_indices_by_cla('A'))}")
```

```
Length of task 0 training data: 66, number of samples for class A: 18
Length of task 0 training data: 66, number of samples for class A: 30
```

The second difference is the concept of incomplete information, so typically in the IIRC setup, the model is not told all the labels of a sample, but only the labels that correspond to the current task, and the model should figure the other labels on its own.

This only applies to the training set though, as for the test set you need to know all the labels to evaluate the model properly

[21]:
```
labels_ = ["A", "Aa", "Ab"]
iirc_dataset_train.reset()
iirc_dataset_test.reset()
for task_id, label in enumerate(labels_):
    iirc_dataset_train.choose_task(task_id)
    iirc_dataset_test.choose_task(task_id)
    train_sample, train_label1, train_label2 = \
        iirc_dataset_train[iirc_dataset_train.get_image_indices_by_cla(label, num_
↪samples=1)[0]]
    test_sample, test_label1, test_label2 = \
        iirc_dataset_test[iirc_dataset_test.get_image_indices_by_cla(label, num_
↪samples=1)[0]]

    print(f"task {task_id}:\ntraining sample label: {(train_label1, train_label2)}")
    print(f"test sample label: {test_label1, test_label2}\n")
```

```
task 0:
training sample label: ('A', 'None')
test sample label: ('A', 'None')
```

(continues on next page)

(continued from previous page)

```
task 1:
training sample label: ('Aa', 'None')
test sample label: ('A', 'Aa')

task 2:
training sample label: ('Ab', 'None')
test sample label: ('A', 'Ab')
```

To make a dataset use the *complete information* mode irrespective of whether it is a training set or a test set, the argument *complete_information_mode* can be provided when initializing the dataset, and the functions *enable_complete_information_mode()* and *enable_incomplete_information_mode()* can be used as well after initialization.

Take note that the *load_tasks_up_to(task_id)* function doesn't work in the *incomplete information*

2.2 Loading IIRC and incremental datasets

2.2.1 Usage with incremental-CIFAR100, IIRC-CIFAR100, incremental-Imagenet, and IIRC-Imagenet

[1]:
```
import sys
sys.path.append("../..")

from iirc.datasets_loader import get_lifelong_datasets
from iirc.definitions import PYTORCH, IIRC_SETUP
from iirc.utils.download_cifar import download_extract_cifar100
```

For using these datasets with the preset tasks schedules, the original *CIFAR100* and/or *ImageNet2012* need to be downloaded first.

In the case of *CIFAR100*, the dataset can be downloaded using the following method

[2]:
```
download_extract_cifar100("../../data")
```

```
downloading CIFAR 100
dataset downloaded
extracting CIFAR 100
dataset extracted
```

In the case of *ImageNet*, it has to be downloaded manually, and be arranged in the following manner: * Imagenet * train * n01440764 * n01443537 * . . . * val * n01440764 * n01443537 * . . .

Then the *get_lifelong_datasets* function should be used. The tasks schedules/configurations preset per dataset are:

- *Incremental-CIFAR100*: 10 configurations, each starting with 50 classes in the first task, followed by 10 tasks each having 5 classes
- *IIRC-CIFAR100*: 10 configurations, each starting with 10 superclasses in the first task, followed by 21 tasks each having 5 classes
- *Incremental-Imagenet-full*: 5 configurations, each starting with 160 classes in the first task, followed by 28 tasks each having 30 classes
- *Incremental-Imagenet-lite*: 5 configurations, each starting with 160 classes in the first task, followed by 9 tasks each having 30 classes

- *IIRC-Imagenet-full*: 5 configurations, each starting with 63 superclasses in the first task, followed by 34 tasks each having 30 classes
- *IIRC-Imagenet-lite*: 5 configurations, each starting with 63 superclasses in the first task, followed by 9 tasks each having 30 classes

Although these configurations might seem they are limiting the choices, but the point here is to have a standard set of tasks and class orders so that the results are comparable across different works, otherwise if needed, new task configurations can be added manually as well in the *metadata* folder

We also need a transformations function that takes the image and converts it to a tensor, as well as normalize the image, apply augmentations, etc.

There are two such functions that can be provided: *essential_transforms_fn* and *augmentation_transforms_fn*

essential_transforms_fn should include any essential transformations that should be applied to the PIL image (such as convert to tensor), while *augmentation_transforms_fn* should also include the essential transformations, in addition to any augmentations that need to be applied (such as random horizontal flipping, etc)

[3]:
```
import torchvision.transforms as transforms

essential_transforms_fn = transforms.ToTensor()
augmentation_transforms_fn = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()
])
```

[4]:
```
# The datasets supported are ("incremental_cifar100", "iirc_cifar100", "incremental_
↪imagenet_full", "incremental_imagenet_lite",
# "iirc_imagenet_full", "iirc_imagenet_lite")
lifelong_datasets, tasks, class_names_to_idx = \
    get_lifelong_datasets(dataset_name = "iirc_cifar100",
                          dataset_root = "../../data", # the imagenet folder (where␣
↪the train and val folders reside, or the parent directory of cifar-100-python folder
                          setup = IIRC_SETUP,
                          framework = PYTORCH,
                          tasks_configuration_id = 0,
                          essential_transforms_fn = essential_transforms_fn,
                          augmentation_transforms_fn = augmentation_transforms_fn,
                          joint = False
                         )
```

```
Creating iirc_cifar100
Setup used: IIRC
Using PyTorch
Dataset created
```

joint can also be set to *True* in case of joint training (all classes will come in one task)

The result of the previous function has the following form:

[5]:
```
lifelong_datasets # four splits
```

[5]:
```
{'train': <iirc.lifelong_dataset.torch_dataset.Dataset at 0x20f7746a670>,
 'intask_valid': <iirc.lifelong_dataset.torch_dataset.Dataset at 0x20f7567bf70>,
 'posttask_valid': <iirc.lifelong_dataset.torch_dataset.Dataset at 0x20f7567bfa0>,
 'test': <iirc.lifelong_dataset.torch_dataset.Dataset at 0x20f7567bfd0>}
```

[6]:
```
print(tasks[:3])
```

```
[['flowers', 'small_mammals', 'trees', 'aquatic_mammals', 'fruit_and_vegetables',
↪'people', 'food_containers', 'vehicles', 'large_carnivores', 'insects'], [
↪'television', 'spider', 'shrew', 'mountain', 'hamster'], ['road', 'poppy',
↪'household_furniture', 'woman', 'bee']]
```

[7]:
```
print(class_names_to_idx)
```

```
{'flowers': 0, 'small_mammals': 1, 'trees': 2, 'aquatic_mammals': 3, 'fruit_and_
↪vegetables': 4, 'people': 5, 'food_containers': 6, 'vehicles': 7, 'large_carnivores
↪': 8, 'insects': 9, 'television': 10, 'spider': 11, 'shrew': 12, 'mountain': 13,
↪'hamster': 14, 'road': 15, 'poppy': 16, 'household_furniture': 17, 'woman': 18, 'bee
↪': 19, 'tulip': 20, 'clock': 21, 'orange': 22, 'beaver': 23, 'rocket': 24, 'bicycle
↪': 25, 'can': 26, 'squirrel': 27, 'wardrobe': 28, 'bus': 29, 'whale': 30, 'sweet_
↪pepper': 31, 'telephone': 32, 'leopard': 33, 'bowl': 34, 'skyscraper': 35, 'baby':␣
↪36, 'cockroach': 37, 'boy': 38, 'lobster': 39, 'motorcycle': 40, 'forest': 41, 'tank
↪': 42, 'orchid': 43, 'chair': 44, 'crab': 45, 'girl': 46, 'keyboard': 47, 'otter':␣
↪48, 'bed': 49, 'butterfly': 50, 'lawn_mower': 51, 'snail': 52, 'caterpillar': 53,
↪'wolf': 54, 'pear': 55, 'tiger': 56, 'pickup_truck': 57, 'cup': 58, 'reptiles': 59,
↪'train': 60, 'sunflower': 61, 'beetle': 62, 'apple': 63, 'palm_tree': 64, 'plain':␣
↪65, 'large_omnivores_and_herbivores': 66, 'rose': 67, 'tractor': 68, 'crocodile':␣
↪69, 'mushroom': 70, 'couch': 71, 'lamp': 72, 'mouse': 73, 'bridge': 74, 'turtle':␣
↪75, 'willow_tree': 76, 'man': 77, 'lizard': 78, 'maple_tree': 79, 'lion': 80,
↪'elephant': 81, 'seal': 82, 'sea': 83, 'dinosaur': 84, 'worm': 85, 'bear': 86,
↪'castle': 87, 'plate': 88, 'dolphin': 89, 'medium_sized_mammals': 90, 'streetcar':␣
↪91, 'bottle': 92, 'kangaroo': 93, 'snake': 94, 'house': 95, 'chimpanzee': 96,
↪'raccoon': 97, 'porcupine': 98, 'oak_tree': 99, 'pine_tree': 100, 'possum': 101,
↪'skunk': 102, 'fish': 103, 'fox': 104, 'cattle': 105, 'ray': 106, 'aquarium_fish':␣
↪107, 'cloud': 108, 'flatfish': 109, 'rabbit': 110, 'trout': 111, 'camel': 112,
↪'table': 113, 'shark': 114}
```

lifelong_datasets has four splits, where *train* is for training, *intask_valid* is for validation during task training (in case of IIRC setup, this split is using *incomplete information* like the *train* split), *posttask_valid* is for validation after each task training (in case of IIRC setup, this split is using *complete information* like the *test* split), and finally the *test* split

[ ]:
```
```

2.3 Lifelong Learning Methods Guide

The lifelong learning methods in this package follow the following procedures

```
example_model = lifelong_methods.methods.example.Model(args)  # replace example with␣
↪whatever module is there

for task in tasks:
    task_data <- load here the task data

    # This method initializes anything that needs to be inizialized at the beginning␣
↪of each task
    example_model.prepare_model_for_new_task(task_data, **kwargs)

    for epoch in epochs:
        # Training
        for minibatch in task_data:
```

(continues on next page)

(continued from previous page)

```
            # This is where the training happens
            predictions, loss = example_model.observe(minibatch)

        # This is where anything that needs to be done after each epoch should be␣
↪done, if any
        example_model.consolidate_epoch_knowledge(**kwargs)

    # This is where anything that needs to be done after the task is done takes place
    example_model.consolidate_task_knowledge(**kwargs)

    # Inference
    # This is where the inference happens
    predictions = example_model(inference_data_batch)
```

This is the typical order of how things flow in a lifelong learning scenario, and how does this package handles that. This order makes it easy to implement new methods with shared base, so that they can run using the same code and experimenting can be fast

When defining a new lifelong learning model, the first step is to create a model that inherits from *lifelong_methods.methods.base_method.BaseMethod*, then the following abstract methods need to be defined (see the methods docs for more details), private methods here are run from inside their similar but public methods so that shared stuff between the different methods doesn't need to be reimplemented (like resetting the scheduler after each task, etc), see the docs to know what is already implemented in the public methods so that you don't reimplement them:

- *_prepare_model_for_new_task*: This private method is run from inside the *prepare_model_for_new_task* in the *BaseMethod*,
- *_consolidate_epoch_knowledge*: This private method is run from inside the *consolidate_epoch_knowledge* in the *BaseMethod*
- *observe*
- *forward*
- *consolidate_task_knowledge*

CHAPTER

THREE

DOCUMENTATION:

3.1 iirc package

3.1.1 Subpackages

iirc.lifelong_dataset package

Submodules

iirc.lifelong_dataset.base_dataset module

class iirc.lifelong_dataset.base_dataset.**BaseDataset**(*dataset: Union[List[Tuple[PIL.Image.Image, Tuple[str, ...]]], List[Tuple[str, Tuple[str, ...]]]], tasks: List[List[str]], setup: str = 'IIRC', using_image_path: bool = False, cache_images: bool = False, essential_transforms_fn: Optional[Callable[[Any], Any]] = None, augmentation_transforms_fn: Optional[Callable[[Any], Any]] = None, test_mode: bool = False, complete_information_mode: Optional[bool] = None, superclass_data_pct: float = 0.6, subclass_data_pct: float = 0.6, superclass_sampling_size_cap: int = 100*)

Bases: abc.ABC

A lifelong learning dataset base class with the underlying data changing based on what task is currently activated. This class is an abstract base class.

Parameters

- **dataset** (*DatasetStructType*) – a list of tuples which contains the data in the form of (image, (label,)) or (image, (label1,label2)). The image path (str) can be provided instead if the images would be loaded on the fly (see the argument using_image_path). label is a string representing the class name

- **tasks** (*List[List[str]]*) – a list of lists where each inner list contains the set of classes (class names) that will be introduced in that task (example: [[dog, cat, car], [tiger, truck, fish]])
- **setup** (*str*) – Class Incremental Learning setup (CIL) or Incremental Implicitly Refined Classification setup (IIRC) (default: IIRC_SETUP)
- **using_image_path** (*bool*) – whether the pillow image is provided in the dataset argument, or the image path that would be used later to load the image. set True if using the image path (default: False)
- **cache_images** (*bool*) – cache images that belong to the current task in the memory, only applicable when using the image path (default: False)
- **essential_transforms_fn** (*Callable[[Any], Any]*) – A function that contains the essential transforms (for example, converting a pillow image to a tensor) that should be applied to each image. This function is applied only when the augmentation_transforms_fn is set to None (as in the case of a test set) or inside the disable_augmentations context (default: None)
- **augmentation_transforms_fn** – (Callable[[Any], Any]): A function that contains the essential transforms (for example, converting a pillow image to a tensor) and augmentation transforms (for example, applying random cropping) that should be applied to each image. When this function is provided, essential_transforms_fn is not used except inside the disable_augmentations context (default: None)
- **test_mode** (*bool*) – Whether this dataset is considered a training split or a test split. This info is only helpful when using the IIRC setup (default: False)
- **complete_information_mode** (*bool*) – Whether the dataset is in complete information mode or incomplete information mode. This is only valid when using the IIRC setup. In the incomplete information mode, if a sample has two labels corresponding to a previous task and a current task (example: dog and Bulldog), only the label present in the current task is provided (Bulldog). In the complete information mode, both labels will be provided. In all cases, no label from a future task would be provided. When no value is set for complete_information_mode, this value is defaulted to the test_mode value (complete information during test mode only) (default: None)
- **superclass_data_pct** (*float*) – The percentage of samples sampled for each superclass from its consistuent subclasses. This is valid only when using the IIRC setup and when test_mode is set to False. For example, If the superclass "dog" has the subclasses "Bulldog" and "Whippet", and superclass_data_pct is set to 0.4, then 40% of each of the "Bulldog" samples and "Whippet" samples will be provided when training on the task that has the class "dog" (default: 0.6)
- **subclass_data_pct** (*float*) – The percentage of samples sampled for each subclass if it has a superclass. This is valid only when using the IIRC setup and when test_mode is set to False. For example, If the superclass "dog" has one of the subclasses as "Bulldog", and superclass_data_pct is set to 0.4 while subclass_data_pct is set to 0.8, then 40% of the "Bulldog" samples will be provided when training on the task that contains "dog", and 80% of the "Bulldog" samples will be provided when training on the task that contains "Bulldog". superclass_data_pct and subclass_data_pct don't need to sum to 1 as the samples can be repeated across tasks (in the previous example, 20% of the samples were repeated across the two tasks) (default: 0.6)
- **superclass_sampling_size_cap** (*int*) – The number of subclasses a superclass should contain after which the number of samples doesn't increase anymore. This is valid only when using the IIRC setup and when test_mode is set to False. For example, If a superclass has 8 subclasses, with the superclass_data_pct set to 0.4, and super-

class_sampling_size_cap set to 5, then superclass_data_pct for that specific superclass will be adjusted to 0.25 (5 / 8 * 0.4) (default: 100)

dataset_state_dict() → Dict
This function returns a dict that contains the current state of the dataset

Returns a dictionary with all the attributes (key is attribute name) and their values, except the attributes in the self.non_savable_attributes

Return type Dict

load_dataset_state_dict(*state_dict: Dict*) → None
This function loads the object attributes with the values in state_dict

Parameters **state_dict** (*Dict*) – a dictionary with the attribute names as keys and their values

reset() → None
Reset the dataset to the starting state

choose_task(*task_id: int*) → None
Load the data corresponding to task "task_id" and update tbe seen classes based on it.

Parameters **task_id** (*int*) – The task_id of the task to load

load_tasks_up_to(*task_id: int*) → None

Load the data corresponding to the tasks up to "task_id" (including that task). When using the IIRC setup, this function is only available when complete_information_mode is set to True.

Parameters **task_id** (*int*) – The task_id of the task to load

get_labels(*index: int*) → Tuple[str, str]
Return the labels of the sample with index (index) in the current task.

Parameters **index** (*int*) – The index of the sample in the current task, this is a relative index within the current task

Returns The labels corresponding to the sample. If using CIL setup, or if the other label is masked, then the other str contains the value specified by the NO_LABEL_PLACEHOLDER

Return type Tuple[str, str]

get_item(*index: int*) → Tuple[Any, str, str]

Return the image with index (index) in the current task along with its labels. No transformations are applied to the image.

Parameters **index** (*int*) – The index of the sample in the current task, this is a relative index within the current task

Returns The image along with its labels . If using CIL setup, or if the other label is masked, then the other str contains the value specified by the NO_LABEL_PLACEHOLDER

Return type Tuple[Any, str, str]

get_image_indices_by_cla(*cla: str*, *num_samples: int = - 1*, *shuffle: bool = True*) → numpy.ndarray

get the indices of the samples of cla within the cur_task. Warning: if the task data is changed (like by using choose_task() or load_tasks_up_to()), these indices would point to other samples as they are relative to the current task

Parameters

- **cla** (*str*) – The class name
- **num_samples** (*int*) – The number of samples needed for that class, set to -1 to return the indices of all the samples that belong to that class in the current task (default: -1)
- **shuffle** (*bool*) – Whether to return the indices shuffled (default: False)

Returns The indices of the samples of class cla within the current task (relative indices)

Return type np.ndarray

disable_augmentations() → None
A context where only the essential transformations are applied

enable_complete_information_mode() → None

enable_incomplete_information_mode() → None

iirc.lifelong_dataset.tensorflow_dataset module

iirc.lifelong_dataset.torch_dataset module

class iirc.lifelong_dataset.torch_dataset.**Dataset**(*dataset: Union[List[Tuple[PIL.Image.Image, Tuple[str, ...]]], List[Tuple[str, Tuple[str, ...]]]], tasks: List[List[str]], setup: str = 'IIRC', using_image_path: bool = False, cache_images: bool = False, essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, test_mode: bool = False, complete_information_mode: Optional[bool] = None, superclass_data_pct: float = 0.6, subclass_data_pct: float = 0.6, superclass_sampling_size_cap: int = 100*)

Bases: *iirc.lifelong_dataset.base_dataset.BaseDataset*, torch.utils.data.dataset.Dataset

A class inhereting from BaseDataset to be used with PyTorch

Parameters

- **dataset** (*DatasetStructType*) – a list of tuples which contains the data in the form of (image, (label,)) or (image, (label1,label2)). The image path (str) can be provided instead if the images would be loaded on the fly (see the argument using_image_path). label is a string representing the class name

- **tasks** (*List[List[str]]*) – a list of lists where each inner list contains the set of classes (class names) that will be introduced in that task (example: [[dog, cat, car], [tiger, truck, fish]])
- **setup** (*str*) – Class Incremental Learning setup (CIL) or Incremental Implicitly Refined Classification setup (IIRC) (default: IIRC_SETUP)
- **using_image_path** (*bool*) – whether the pillow image is provided in the dataset argument, or the image path that would be used later to load the image. set True if using the image path (default: False)
- **cache_images** (*bool*) – cache images that belong to the current task in the memory, only applicable when using the image path (default: False)
- **essential_transforms_fn** (*Optional[Callable[[Image.Image], torch.Tensor]]*) – A function that contains the essential transforms (for example, converting a pillow image to a tensor) that should be applied to each image. This function is applied only when the augmentation_transforms_fn is set to None (as in the case of a test set) or inside the disable_augmentations context (default: None)
- **augmentation_transforms_fn** – (Optional[Callable[[Image.Image], torch.Tensor]]): A function that contains the essential transforms (for example, converting a pillow image to a tensor) and augmentation transforms (for example, applying random cropping) that should be applied to each image. When this function is provided, essential_transforms_fn is not used except inside the disable_augmentations context (default: None)
- **test_mode** (*bool*) – Whether this dataset is considered a training split or a test split. This info is only helpful when using the IIRC setup (default: False)
- **complete_information_mode** (*bool*) – Whether the dataset is in complete information mode or incomplete information mode. This is only valid when using the IIRC setup. In the incomplete information mode, if a sample has two labels corresponding to a previous task and a current task (example: dog and Bulldog), only the label present in the current task is provided (Bulldog). In the complete information mode, both labels will be provided. In all cases, no label from a future task would be provided. When no value is set for complete_information_mode, this value is defaulted to the test_mode value (complete information during test mode only) (default: None)
- **superclass_data_pct** (*float*) – The percentage of samples sampled for each superclass from its consistuent subclasses. This is valid only when using the IIRC setup and when test_mode is set to False. For example, If the superclass "dog" has the subclasses "Bulldog" and "Whippet", and superclass_data_pct is set to 0.4, then 40% of each of the "Bulldog" samples and "Whippet" samples will be provided when training on the task that has the class "dog" (default: 0.6)
- **subclass_data_pct** (*float*) – The percentage of samples sampled for each subclass if it has a superclass. This is valid only when using the IIRC setup and when test_mode is set to False. For example, If the superclass "dog" has one of the subclasses as "Bulldog", and superclass_data_pct is set to 0.4 while subclass_data_pct is set to 0.8, then 40% of the "Bulldog" samples will be provided when training on the task that contains "dog", and 80% of the "Bulldog" samples will be provided when training on the task that contains "Bulldog". superclass_data_pct and subclass_data_pct don't need to sum to 1 as the samples can be repeated across tasks (in the previous example, 20% of the samples were repeated across the two tasks) (default: 0.6)
- **superclass_sampling_size_cap** (*int*) – The number of subclasses a superclass should contain after which the number of samples doesn't increase anymore. This is

valid only when using the IIRC setup and when test_mode is set to False. For example, If a superclass has 8 subclasses, with the superclass_data_pct set to 0.4, and superclass_sampling_size_cap set to 5, then superclass_data_pct for that specific superclass will be adjusted to 0.25 (5 / 8 * 0.4) (default: 100)

Module contents

iirc.utils package

Submodules

iirc.utils.download_cifar module

iirc.utils.download_cifar.**download_extract_cifar100**(*root='data'*)

iirc.utils.prepare_imagenet module

iirc.utils.utils module

iirc.utils.utils.**print_msg**(*msg*)

iirc.utils.utils.**unpickle**(*file*)

Module contents

3.1.2 Submodules

3.1.3 iirc.datasets_loader module

iirc.datasets_loader.**get_lifelong_datasets**(*dataset_name: str*, *dataset_root: str = './data'*, *setup: str = 'IIRC'*, *framework: str = 'PyTorch'*, *tasks_configuration_id: int = 0*, *essential_transforms_fn: Optional[Callable[[PIL.Image.Image], Any]] = None*, *augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], Any]] = None*, *cache_images: bool = False*, *joint: bool = False*) → Tuple[Dict[str, *iirc.lifelong_dataset.base_dataset.BaseDataset*], List[List[str]], Dict[str, int]]

Get the incremental refinement learning , as well as the tasks (which contains the classes introduced at each task), and the index for each class corresponding to its order of appearance

Parameters

- **dataset_name** (*str*) – The name of the dataset, ex: iirc_cifar100
- **dataset_root** (*str*) – The directory where the dataset is/will be downloaded (default: "./data")
- **setup** (*str*) – Class Incremental Learning setup (CIL) or Incremental Implicitly Refined Classification setup (IIRC) (default: IIRC_SETUP)

- **framework** (*str*) – The framework to be used, whether PyTorch or Tensorflow. use Tensorflow for any numpy based dataloading (default: PYTORCH)
- **tasks_configuration_id** (*int*) – The configuration id, where each configuration corresponds to a specific tasks and classes order for each dataset. This id starts from 0 for each dataset. Ignore when joint is set to True (default: 0)
- **essential_transforms_fn** (*Optional[Callable[[Image.Image], Any]]*) – A function that contains the essential transforms (for example, converting a pillow image to a tensor) that should be applied to each image. This function is applied only when the augmentation_transforms_fn is set to None (as in the case of a test set) or inside the disable_augmentations context (default: None)
- **augmentation_transforms_fn** – A function that contains the essential transforms (for example, converting a pillow image to a tensor) and augmentation transforms (for example, applying random cropping) that should be applied to each image. When this function is provided, essential_transforms_fn is not used except inside the disable_augmentations context (default: None)
- **cache_images** (*bool*) – cache images that belong to the current task in the memory, only applicable when using the image path (default: False)
- **joint** (*bool*) – provided all the classes in a single task for joint training (default: False)

Returns

lifelong_datasets (Dict[str, BaseDataset]): a dictionary with the keys corresponding to the four splits (train, intask_validation, posttask_validation, test), and the values containing the dataset object inheriting from BaseDataset for that split.

tasks (List[List[str]]): a list of lists where each inner list contains the set of classes (class names) that will be introduced in that task (example: [[dog, cat, car], [tiger, truck, fish]]).

class_names_to_idx (Dict[str, int]): a dictionary with the class name as key, and the class index as value (example: {"dog": 0, "cat": 1, . . . }).

Return type Tuple[Dict[str, *BaseDataset*], List[List[str]], Dict[str, int]]

3.1.4 iirc.definitions module

3.1.5 Module contents

3.2 lifelong_methods package

3.2.1 Subpackages

lifelong_methods.buffer package

Submodules

lifelong_methods.buffer.buffer module

class lifelong_methods.buffer.buffer.**BufferBase**(*config: Dict*, *buffer_dir: Optional[str] = None*, *map_size: int = 1000000000.0*, *essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*, *augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: abc.ABC, torch.utils.data.dataset.Dataset

A buffer that saves memories from current task to replay them during later tasks

Parameters

- **config** (*Dict*) – A dictionary that has the following key value pairs: n_memories_per_class (int): Number of memories/samples to save per class, set to -1 to use total_n_mems total_n_mems (int): The total number of memories to save (mutually exclusive with n_memories_per_class) max_mems_pool_per_class (int): The pool size per class to sample randomly the memories from which the buffer

  chooses what memories to keep, set to -1 to choose memories from all the class samples
- **buffer_dir** (*Optional[str]*) – The directory where the buffer data will be kept (None for keeping the buffer data in memory) (default: None)
- **map_size** (*int*) – Th estimated size of the buffer lmdb database, in bytes (defalt: 1e9)
- **essential_transforms_fn** (*Optional[Callable[[Image.Image], torch.Tensor]]*) – A function that contains the essential transforms (for example, converting a pillow image to a tensor) that should be applied to each image. This function is applied only when the augmentation_transforms_fn is set to None (as in the case of a test set) or inside the disable_augmentations context (default: None)
- **augmentation_transforms_fn** – (Optional[Callable[[Image.Image], torch.Tensor]]): A function that contains the essential transforms (for example, converting a pillow image to a tensor) and augmentation transforms (for example, applying random cropping) that should be applied to each image. When this function is provided, essential_transforms_fn is not used except inside the disable_augmentations context (default: None)

buffer_state_dict() → Dict

This function returns a dict that contains the current state of the buffer

Returns a dictionary with all the attributes (key is attribute name) and their values, except the attributes in the self.non_savable_attributes

Return type Dict

load_buffer_state_dict(*state_dict: Dict*) → None

This function loads the object attributes with the values in state_dict

Parameters **state_dict** (*Dict*) – a dictionary with the attribute names as keys and their values

get_image_indices_by_class(*class_name: str*) → numpy.ndarray

get the indices of the samples of class "class_name"

Parameters **class_name** (*str*) – The class name

Returns The indices of the samples of class "class_name"

Return type np.ndarray

begin_adding_samples_to_lmdb() → None

A function that needs to be called before adding samples to the buffer, in case of using an lmdb buffer, so that a transaction is created.

end_adding_samples_to_lmdb() → None

A function that needs to be called after adding samples to the buffer is done, in case of using an lmdb buffer, so that the transaction is committed.

reset_lmdb_database() → None

A function that needs to be called after each epoch, in case of using an lmdb dataset, to close the environment and open a new one to kill active readers

add_sample(*class_label: str*, *image: PIL.Image.Image*, *labels: Tuple[str, str]*, *rank: int = 0*) → None
Add a sample to the buffer.

Parameters

- **class_label** (*str*) – The class label of the image, and in case the image has multiple labels, the class label for which the sample should be associated with in the buffer
- **image** (*Image.Image*) – The image to be added
- **labels** (*Tuple[str, str]*) – The labels of the image (including the class_label), in case the image has only one label, provide the second label as NO_LABEL_PLACEHOLDER
- **rank** (*int*) – The rank of the current gpu, in case of using multiple gpus

remove_samples(*class_label: str*, *n: int*) → None
Remove a number (n) of the samples associated with class "class_label".

Parameters

- **class_label** (*str*) – The class label of which the sample is associated with in the buffer
- **n** (*int*) – The number of samples to remove

update_buffer_new_task(*new_task_data:* iirc.lifelong_dataset.torch_dataset.Dataset, *dist_args: Optional[Dict] = None*, ***kwargs*) → None

Update the buffer by adding samples of classes of a new task, after removing samples associated with the older classes in case the buffer has a fixed size (self.fixed_n_mems_per_cla is set to False)

Parameters

- **new_task_data** (Dataset) – The new task data
- **dist_args** (*Optional[Dict]*) – a dictionary of the distributed processing values in case of multiple gpu (ex:
- **of the device) (default** (*rank*) – None)
- ****kwargs** – arguments associated with each method

disable_augmentations()
A context where only the essential transformations are applied

class lifelong_methods.buffer.buffer.**TaskDataMergedWithBuffer**(*buffer: lifelong_methods.buffer.buffer.BufferBase*, *task_data: iirc.lifelong_dataset.torch_dataset.Dataset*, *buffer_sampling_multiplier: float = 1.0*)

Bases: torch.utils.data.dataset.Dataset

A torch dataset object that merges the task data and the buffer with the specified options

Parameters

- **buffer** (BufferBase) – A buffer object that includes the memories from previous classes
- **task_data** (*data.Dataset*) – A dataset object that contains the new task data
- **buffer_sampling_multiplier** (*float*) – A multiplier for sampling from the buffer more/less times than the size of the buffer (for example a multiplier of 2 samples from the buffer (with replacement) twice its size per epoch, a multiplier of 1 ensures that all the buffer samples will be retrieved once")

Module contents

lifelong_methods.methods package

Submodules

lifelong_methods.methods.agem module

class lifelong_methods.methods.agem.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]]*, *class_names_to_idx: Dict[str, int]*, *config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

An implementation of A-GEM from Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient Lifelong Learning with A-GEM. ICLR, 2019.

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor
The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None
Takes place after training on each task

class lifelong_methods.methods.agem.**Buffer**(*config: Dict, buffer_dir: Optional[str] = None, map_size: int = 1000000000.0, essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)
Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.base_method module

class lifelong_methods.methods.base_method.**BaseMethod**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]], class_names_to_idx: Dict[str, int], config: Dict*)
Bases: abc.ABC, torch.nn.modules.module.Module

A base model for all the lifelong learning methods to inherit, which contains all common functionality

Parameters

- **n_cla_per_tsk** (*Union[np.ndarray, List[int]]*) – An integer numpy array including the number of classes per each task.
- **class_names_to_idx** (*Dict[str, int]*) – The index of each class name
- **config** (*Dict*) – A dictionary that has the following key value pairs: temperature (float): the temperature to divide the logits by memory_strength (float): The weight to add for the samples from the buffer when computing the loss

  (not implemented yet)

  n_layers (int): The number of layers for the network used (not all values are allowed depending on the architecture)

  dataset (str): The name of the dataset (for ex: iirc_cifar100) optimizer (str): The type of optimizer (“momentum” or “adam”) lr (float): The initial learning rate lr_schedule (Optional[list[int]]): The epochs for which the learning rate changes lr_gamma (float): The multiplier multiplied by the learning rate at the epochs specified in lr_schedule reduce_lr_on_plateau (bool): reduce learning rate on plateau weight_decay (float): the weight decay multiplier

method_state_dict() → Dict[str, Dict]

This function returns a dict that contains the state dictionaries of this method (including the model, the optimizer, the scheduler, as well as the values of the variables whose names are inside the self.method_variables), so that they can be used for checkpointing.

Returns a dictionary with the state dictionaries of this method, the optimizer, the scheduler, and the values of the variables whose names are inside the self.method_variables

Return type Dict

load_method_state_dict(*state_dicts: Dict[str, Dict]*) → None

This function loads the state dicts of the various parts of this method (along with the variables in self.method_variables)

Parameters

- **state_dicts** (*Dict[str, Dict]*) – a dictionary with the state dictionaries of this method, the optimizer, the
- **scheduler** –
- **the values of the variables whose names are inside the self.method_variables** (*and*) –

reset_optimizer_and_scheduler(*optimizable_parameters: Optional[Iterator[torch.nn.parameter.Parameter]] = None*) → None

Reset the optimizer and scheduler after a task is done (with the option to specify which parameters to optimize

Parameters **(Optional[Iterator[nn.parameter.Parameter]]** (*optimizable_parameters*) – specify the parameters that should be optimized, in case some parameters needs to be frozen (default: None)

get_last_lr() → List[float]

Get the current learning rate

step_scheduler(*val_metric: Optional = None*) → None

Take a step with the scheduler (should be called after each epoch)

Parameters **val_metric** (*Optional*) – a metric to compare in case of reducing the learning rate on plateau (default: None)

forward_net(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

an alias for self.net(x)

Parameters **x** (*torch.Tensor*) – The batch of images

Returns output (torch.Tensor): The network output of shape (minibatch size x output size) latent (torch.Tensor): The network latent variable of shape (minibatch size x last hidden size)

Return type Tuple[torch.Tensor, torch.Tensor]

prepare_model_for_new_task(*task_data: Optional[*iirc.lifelong_dataset.torch_dataset.Dataset*] = None*, *dist_args: Optional[dict] = None*, ***kwargs*) → None

Takes place before the starting epoch of each new task.

The shared functionality among the methods is that the seen classes are updated and the optimizer and scheduler are reset. (see _prepare_model_for_new_task for method specific functionality)

Parameters

- **task_data** (*Optional[Dataset]*) – The new task data (default: None)

- **dist_args** (*Optional[Dict]*) – a dictionary of the distributed processing values in case of multiple gpu (ex:
- **of the device) (default** (*rank*) – None)
- ****kwargs** – arguments that are method specific

consolidate_epoch_knowledge(*val_metric=None*, ***kwargs*) → None
Takes place after training on each epoch

The shared functionality among the methods is that the scheduler takes a step. (see _consolidate_epoch_knowledge for method specific functionality)

Parameters

- **val_metric** (*Optional*) – a metric to compare in case of reducing the learning rate on plateau (default: None)
- ****kwargs** – arguments that are method specific

abstract observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]
The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

abstract consolidate_task_knowledge(***kwargs*) → None
Takes place after training each task This function needs to be defined in the inheriting method class

Parameters ****kwargs** – arguments that are method specific

abstract forward(*x: torch.Tensor*) → torch.Tensor
The method used during inference, returns a tensor of model predictions This function needs to be defined in the inheriting method class

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

lifelong_methods.methods.finetune module

class lifelong_methods.methods.finetune.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]], class_names_to_idx: Dict[str, int], config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

A finetuning (Experience Replay) baseline.

observe(*x: torch.Tensor, y: torch.Tensor, in_buffer: Optional[torch.Tensor] = None, train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None

Takes place after training on each task

class lifelong_methods.methods.finetune.**Buffer**(*config: Dict, buffer_dir: Optional[str] = None, map_size: int = 1000000000.0, essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.icarl module

class lifelong_methods.methods.icarl.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]], class_names_to_idx: Dict[str, int], config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

An implementation of iCaRL from S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. iCaRL: Incremental classifier and representation learning. CVPR, 2017.

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions Classification is done during inference using the nearest class mean

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(*buffer: lifelong_methods.buffer.buffer.BufferBase*, *device: torch.device*, *batch_size: int*, ***kwargs*) → None

Takes place after training each task. It updates the class means based on the new set of exemplars

Parameters

- **buffer** (*BufferBase*) – The replay buffer
- **device** (*torch.device*) – The device used (cpu or gpu)
- **batch_size** (*int*) – The batch size to be used for calculating the class mean

class lifelong_methods.methods.icarl.**Buffer**(*config: Dict, buffer_dir: Optional[str] = None, map_size: int = 1000000000.0, essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.icarl_cnn module

class lifelong_methods.methods.icarl_cnn.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]]*, *class_names_to_idx: Dict[str, int]*, *config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

An implementation of modified version of iCaRL that doesn't use the nearest class mean during inference

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None

Takes place after training on each task

class lifelong_methods.methods.icarl_cnn.**Buffer**(*config: Dict*, *buffer_dir: Optional[str] = None*, *map_size: int = 1000000000.0*, *essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*, *augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.icarl_norm module

class lifelong_methods.methods.icarl_norm.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]]*, *class_names_to_idx: Dict[str, int]*, *config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

An implementation of modified version of iCaRL that doesn't use the nearest class mean during inference, and the the output layer with a cosine similarity layer (where the weights and features are normalized before applying the dot product)

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None

Takes place after training each task

class lifelong_methods.methods.icarl_norm.**Buffer**(*config: Dict*, *buffer_dir: Optional[str] = None*, *map_size: int = 1000000000.0*, *essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*, *augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.lucir module

class lifelong_methods.methods.lucir.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]]*, *class_names_to_idx: Dict[str, int]*, *config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

An implementation of LUCIR from Saihui Hou, Xinyu Pan, Chen Change Loy, Zilei Wang, and Dahua Lin. Learning a Unified Classifier Incrementally via Rebalancing. CVPR, 2019.

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None

Takes place after training on each task

class lifelong_methods.methods.lucir.**Buffer**(*config: Dict*, *buffer_dir: Optional[str] = None*, *map_size: int = 1000000000.0*, *essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*, *augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

lifelong_methods.methods.mask_seen_classes module

class lifelong_methods.methods.mask_seen_classes.**Model**(*n_cla_per_tsk: Union[numpy.ndarray, List[int]], class_names_to_idx: Dict[str, int], config: Dict*)

Bases: *lifelong_methods.methods.base_method.BaseMethod*

A method that only provides feedback for the classes that belong to the task of the sample (masks other tasks classes when computing the loss).

observe(*x: torch.Tensor*, *y: torch.Tensor*, *in_buffer: Optional[torch.Tensor] = None*, *train: bool = True*) → Tuple[torch.Tensor, float]

The method used for training and validation, returns a tensor of model predictions and the loss This function needs to be defined in the inheriting method class

Parameters

- **x** (*torch.Tensor*) – The batch of images
- **y** (*torch.Tensor*) – A 2-d batch indicator tensor of shape (number of samples x number of classes)
- **in_buffer** (*Optional[torch.Tensor]*) – A 1-d boolean tensor which indicates which sample is from the buffer.
- **train** (*bool*) – Whether this is training or validation/test

Returns predictions (torch.Tensor) : a 2-d float tensor of the model predictions of shape (number of samples x number of classes) loss (float): the value of the loss

Return type Tuple[torch.Tensor, float]

forward(*x: torch.Tensor*) → torch.Tensor

The method used during inference, returns a tensor of model predictions

Parameters **x** (*torch.Tensor*) – The batch of images

Returns a 2-d float tensor of the model predictions of shape (number of samples x number of classes)

Return type torch.Tensor

consolidate_task_knowledge(***kwargs*) → None

Takes place after training on each task

class lifelong_methods.methods.mask_seen_classes.**Buffer**(*config: Dict, buffer_dir: Optional[str] = None, map_size: int = 1000000000.0, essential_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None, augmentation_transforms_fn: Optional[Callable[[PIL.Image.Image], torch.Tensor]] = None*)

Bases: *lifelong_methods.buffer.buffer.BufferBase*

Module contents

lifelong_methods.models package

Submodules

lifelong_methods.models.cosine_linear module

Adapted from https://github.com/hshustc/CVPR19_Incremental_Learning/blob/master/cifar100-class-incremental/modified_linear.py

Reference: [1] Saihui Hou, Xinyu Pan, Chen Change Loy, Zilei Wang, Dahua Lin

Learning a Unified Classifier Incrementally via Rebalancing. CVPR 2019

class lifelong_methods.models.cosine_linear.**CosineLinear**(*in_features*, *out_features*, *sigma: Union[bool, float, int] = True*)

Bases: torch.nn.modules.module.Module

reset_parameters()

forward(*input_: torch.Tensor*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class lifelong_methods.models.cosine_linear.**SplitCosineLinear**(*in_features*, *out_features1*, *out_features2*, *sigma: Union[bool, float, int] = True*)

Bases: torch.nn.modules.module.Module

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

lifelong_methods.models.mlp module

class lifelong_methods.models.mlp.**MLP**(*input_shape*, *num_tasks*, *classes_per_task*, *num_hidden_layers=1*, *hidden_sizes=128*, *multi_head=False*)

Bases: torch.nn.modules.module.Module

forward(*input_*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

lifelong_methods.models.resnet module

class lifelong_methods.models.resnet.**ResNet**(*num_classes=10*, *num_layers=18*)

Bases: torch.nn.modules.module.Module

forward(*input_*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

lifelong_methods.models.resnetcifar module

Taken with some modifications from the code written by Yerlan Idelbayev https://github.com/akamaster/pytorch_resnet_cifar10

Properly implemented ResNet-s for CIFAR10 as described in paper [1]. The implementation and structure of this file is hugely influenced by [2] which is implemented for ImageNet and doesn't have option A for identity. Moreover, most of the implementations on the web is copy-paste from torchvision's resnet and has wrong number of params. Proper ResNet-s for CIFAR10 (for fair comparision and etc.) has following number of layers and parameters: name | layers | params ResNet20 | 20 | 0.27M ResNet32 | 32 | 0.46M ResNet44 | 44 | 0.66M ResNet56 | 56 | 0.85M ResNet110 | 110 | 1.7M ResNet1202| 1202 | 19.4m which this implementation indeed has. Reference: [1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Deep Residual Learning for Image Recognition. arXiv:1512.03385

[2] https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

class lifelong_methods.models.resnetcifar.**ResNetCIFAR**(*num_classes=10*, *num_layers=20*, *relu_last_hidden=False*)

Bases: torch.nn.modules.module.Module

forward(*input_*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Module contents

3.2.2 Submodules

3.2.3 lifelong_methods.utils module

class lifelong_methods.utils.**SubsetSampler**(*indices*)

Bases: `torch.utils.data.sampler.Sampler`

Samples elements in order from a given list of indices, without replacement.

Parameters **indices** (*sequence*) – a sequence of indices

lifelong_methods.utils.**get_optimizer**(*model_parameters: Iterator[torch.nn.parameter.Parameter], optimizer_type: str = 'momentum', lr: float = 0.01, lr_gamma: float = 1.0, lr_schedule: Optional[List[int]] = None, reduce_lr_on_plateau: bool = False, weight_decay: float = 0.0001*) → Tuple[torch.optim.optimizer.Optimizer, Union[torch.optim.lr_scheduler.MultiStepLR, torch.optim.lr_scheduler.ReduceLROnPlateau, torch.optim.lr_scheduler.LambdaLR]]

A method that returns the optimizer and scheduler to be used

Parameters

- **model_parameters** (*Iterator[nn.parameter.Parameter]*) – the list of model parameters
- **optimizer_type** (*string*) – the optimizer type to be used (currently only "momentum" and "adam" are supported)
- **lr** (*float*) – The initial learning rate for each task
- **lr_gamma** (*float*) – The multiplicative factor for learning rate decay at the epochs specified
- **lr_schedule** (*Optional[List[int]]*) – the epochs per task at which to multiply the current learning rate by lr_gamma (resets after each task)
- **reduce_lr_on_plateau** (*bool*) – reduce the lr on plateau based on the validation performance metric. If set to True, the lr_schedule is ignored
- **weight_decay** (*float*) – The weight decay multiplier

Returns optimizer (optim.Optimizer): scheduler (Union[MultiStepLR, ReduceLROnPlateau, LambdaLR]):

Return type Tuple[optim.Optimizer, Union[MultiStepLR, ReduceLROnPlateau, LambdaLR]]

lifelong_methods.utils.**labels_index_to_one_hot**(*labels: torch.Tensor, length: int*) → torch.Tensor

lifelong_methods.utils.**l_distance**(*input_vectors: torch.Tensor*, *ref_vectors: torch.Tensor*, *p: Optional[Union[float, str]] = 2*) → torch.Tensor

lifelong_methods.utils.**contrastive_distance_loss**(*input_vectors: torch.Tensor*, *ref_vectors: torch.Tensor*, *labels_one_hot: torch.Tensor*, *p: Optional[Union[float, str]] = 2*, *temperature: float = 1.0*) → Tuple[torch.Tensor, torch.Tensor]

lifelong_methods.utils.**triplet_margin_loss**(*input_vectors: torch.Tensor*, *ref_vectors: torch.Tensor*, *labels_one_hot: torch.Tensor*, *p: Optional[Union[float, str]] = 2*, *base_margin: float = 1*) → Tuple[torch.Tensor, torch.Tensor]

lifelong_methods.utils.**get_gradient**(*model: torch.nn.modules.module.Module*) → torch.Tensor

Get current gradients of a PyTorch model.

This collects ALL GRADIENTS of the model in a SINGLE VECTOR.

lifelong_methods.utils.**update_gradient**(*model: torch.nn.modules.module.Module*, *new_grad: torch.Tensor*) → None

Overwrite current gradient values in Pytorch model. This expects a SINGLE VECTOR containing all corresponding gradients for the model. This means that the number of elements of the vector must match the number of gradients in the model.

lifelong_methods.utils.**transform_labels_names_to_vector**(*labels_names: Iterable[str]*, *num_seen_classes: int*, *class_names_to_idx: Dict[str, int]*) → torch.Tensor

lifelong_methods.utils.**copy_freeze**(*model: torch.nn.modules.module.Module*) → torch.nn.modules.module.Module

Create a copy of the model, with all the parameters frozer (requires_grad set to False)

Parameters **model** (*nn.Module*) – The model that needs to be copied

Returns The frozen model

Return type nn.Module

lifelong_methods.utils.**save_model**(*save_file: str*, *config: Dict*, *metadata: Dict*, *model: Optional[BaseMethod] = None*, *buffer: Optional[BufferBase] = None*, *datasets: Optional[Dict[str, iirc.lifelong_dataset.torch_dataset.Dataset]] = None*, ***kwargs*) → None

Saves the experiment configuration and the state dicts of the model, buffer, datasets, plus any additional data

Parameters

- **save_file** (*str*) – The checkpointing file path
- **config** (*Dict*) – The config of the experiment
- **metadata** (*Dict*) – The metadata of the experiment
- **model** (*Optional[BaseMethod]*) – The Method object (subclass of BaseMethod) for which the state dict should be saved (Default: None)

- **buffer** (*Optional[BufferBase]*) – The buffer object for which the state dict should be saved (Default: None)
- **datasets** (*Optional[Dict[str, Dataset]]*) – The different dataset splits for which the state dict should be saved (Default: None)
- ****kwargs** – Any additional key value pairs that need to be saved

lifelong_methods.utils.**load_model**(*checkpoint: Dict[str, Dict]*, *model: Optional[BaseMethod] = None*, *buffer: Optional[BufferBase] = None*, *datasets: Optional[Dict[str, iirc.lifelong_dataset.torch_dataset.Dataset]] = None*) → None

Loads the state dicts of the model, buffer and datasets

Parameters

- **checkpoint** (*Dict[str, Dict]*) – A dictionary of the state dictionaries
- **model** (*Optional[BaseMethod]*) – The Method object (subclass of BaseMethod) for which the state dict should be updated (Default: None)
- **buffer** (*Optional[BufferBase]*) – The buffer object for which the state dict should be updated (Default: None)
- **datasets** (*Optional[Dict[str, Dataset]]*) – The different dataset splits for which the state dict should be updated (Default: None)

3.2.4 Module contents

- genindex
- modindex
- search

3.3 Paper

IIRC is introduced in the paper "IIRC: Incremental Implicitly-Refined Classification ". If you find this work useful for your research, this is the way to cite it:

```
@misc{abdelsalam2021iirc,
    title = {IIRC: Incremental Implicitly-Refined Classification},
    author={Mohamed Abdelsalam and Mojtaba Faramarzi and Shagun Sodhani and Sarath␣
↪Chandar},
    year={2021}, eprint={2012.12477}, archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

3.4 Community

If you think you can help us make the **iirc** and **lifelong_methods** packages more useful for the lifelong learning community, please don't hesitate to submit an issue or send a pull request.

PYTHON MODULE INDEX

i

l

INDEX

F

G

I

L

M

O

P

R

S

T

U