Flash

PyTorch Lightning

Sep 07, 2021

GETTING STARTED

CHAPTER

ONE

QUICK START

Flash is a high-level deep learning framework for fast prototyping, baselining, finetuning and solving deep learning problems. It features a set of tasks for you to use for inference and finetuning out of the box, and an easy to implement API to customize every step of the process for full flexibility.

Flash is built for beginners with a simple API that requires very little deep learning background, and for data scientists, Kagglers, applied ML practitioners and deep learning researchers that want a quick way to get a deep learning baseline with advanced features PyTorch Lightning offers.

1.1 Why Flash?

1.1.1 For getting started with Deep Learning

Easy to learn

If you are just getting started with deep learning, Flash offers common deep learning tasks you can use out-of-the-box in a few lines of code, no math, fancy nn.Modules or research experience required!

Easy to scale

Flash is built on top of PyTorch Lightning, a powerful deep learning research framework for training models at scale. With the power of Lightning, you can train your flash tasks on any hardware: CPUs, GPUs or TPUs without any code changes.

Easy to upskill

If you want to create more complex and customized models, you can refactor any part of flash with PyTorch or PyTorch Lightning components to get all the flexibility you need. Lightning is just organized PyTorch with the unnecessary engineering details abstracted away.

- Flash (high-level)
- Lightning (mid-level)
- PyTorch (low-level)

When you need more flexibility you can build your own tasks or simply use Lightning directly.

1.1.2 For Deep learning research

Quickest way to a baseline

PyTorch Lightning is designed to abstract away unnecessary boilerplate, while enabling maximal flexibility. In order to provide full flexibility, solving very common deep learning problems such as classification in Lightning still requires some boilerplate. It can still take quite some time to get a baseline model running on a new dataset or out of domain task. We created Flash to answer our users need for a super quick way to baseline for Lightning using proven backbones for common data patterns. Flash aims to be the easiest starting point for your research- start with a Flash Task to benchmark against, and override any part of flash with Lightning or PyTorch components on your way to SOTA research.

Flexibility where you want it

Flash tasks are essentially LightningModules, and the Flash Trainer is a thin wrapper for the Lightning Trainer. You can use your own LightningModule instead of the Flash task, the Lightning Trainer instead of the flash trainer, etc. Flash helps you focus even more only on your research, and less on anything else.

Standard best practices

Flash tasks implement the standard best practices for a variety of different models and domains, to save you time digging through different implementations. Flash abstracts even more details than Lightning, allowing deep learning experts to share their tips and tricks for solving scoped deep learning problems.

Tip: Read *here* to understand when to use Flash vs Lightning.

1.2 Tasks

Flash is comprised of a collection of Tasks. The Flash tasks are laser-focused objects designed to solve a well-defined type of problem, using state-of-the-art methods.

The Flash tasks contain all the relevant information to solve the task at hand- the number of class labels you want to predict, number of columns in your dataset, as well as details on the model architecture used such as loss function, optimizers, etc.

Here are examples of tasks:

```
from flash.text import TextClassifier
from flash.image import ImageClassifier
from flash.tabular import TabularClassifier
```

Note: Tasks are inflexible by definition! To get more flexibility, you can simply use `LightningModule` directly or modify an existing task in just a few lines.

1.3 Inference

Inference is the process of generating predictions from trained models. To use a task for inference:

1. Init your task with pretrained weights using a checkpoint (a checkpoint is simply a file that capture the exact value of all parameters used by a model). Local file or URL works.
2. Pass in the data to *flash.core.model.Task.predict()*.

Here's an example of inference:

```
# import our libraries
from flash.text import TextClassifier

# 1. Init the finetuned task from URL
model = TextClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/text_
↪classification_model.pt")

# 2. Perform inference from list of sequences
predictions = model.predict(
    [
        "Turgid dialogue, feeble characterization - Harvey Keitel a judge?.",
        "The worst movie in the history of cinema.",
        "This guy has done a great job with this movie!",
    ]
)
print(predictions)
```

We get the following output:

```
["negative", "negative", "positive"]
```

1.4 Finetuning

Finetuning (or transfer-learning) is the process of tweaking a model trained on a large dataset, to your particular (likely much smaller) dataset. All Flash tasks have pre-trained backbones that are already trained on large datasets such as ImageNet. Finetuning on pretrained models decreases training time significantly.

To use a Task for finetuning:

1. Load your data and organize it using a DataModule customized for the task (example: `ImageClassificationData`).
2. Choose and initialize your Task which has state-of-the-art backbones built in (example: `ImageClassifier`).
3. Init a *flash.core.trainer.Trainer*.
4. Choose a finetune strategy (example: "freeze") and call *flash.core.trainer.Trainer.finetune()* with your data.

5. Save your finetuned model.

Here's an example of finetuning.

```
from pytorch_lightning import seed_everything

import flash
from flash.core.classification import Labels
from flash.core.data.utils import download_data
from flash.image import ImageClassificationData, ImageClassifier

# set the random seeds.
seed_everything(42)

# 1. Download and organize the data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
)

# 2. Build the model using desired Task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes)

# 3. Create the trainer (run one epoch for demo)
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())

# 4. Finetune the model
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 5. Save the model!
trainer.save_checkpoint("image_classification_model.pt")
```

1.4.1 Using a finetuned model

Once you've finetuned, use the model to predict:

```
# Serialize predictions as labels, automatically inferred from the training data in part␣
↪2.
model.serializer = Labels()

predictions = model.predict(
    [
        "data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg",
        "data/hymenoptera_data/val/ants/2255445811_dabcdf7258.jpg",
    ]
```

(continues on next page)

(continued from previous page)

```
)
print(predictions)
```

We get the following output:

```
['bees', 'ants']
```

Or you can use the saved model for prediction anywhere you want!

```
from flash.image import ImageClassifier

# load finetuned checkpoint
model = ImageClassifier.load_from_checkpoint("image_classification_model.pt")

predictions = model.predict("path/to/your/own/image.png")
```

1.5 Training

When you have enough data, you're likely better off training from scratch instead of finetuning.

To train a task from scratch:

1. Load your data and organize it using a DataModule customized for the task (example: `ImageClassificationData`).
2. Choose and initialize your Task (setting `pretrained=False`) which has state-of-the-art backbones built in (example: `ImageClassifier`).
3. Init a *flash.core.trainer.Trainer* or a `pytorch_lightning.trainer.Trainer`.
4. Call *flash.core.trainer.Trainer.fit()* with your data set.
5. Save your trained model.

Here's an example:

```
from pytorch_lightning import seed_everything

import flash
from flash.core.classification import Labels
from flash.core.data.utils import download_data
from flash.image import ImageClassificationData, ImageClassifier

# set the random seeds.
seed_everything(42)

# 1. Download and organize the data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")
```

(continues on next page)

(continued from previous page)

```
datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
)

# 2. Build the model using desired Task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes,␣
↪pretrained=False)

# 3. Create the trainer (run one epoch for demo)
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())

# 4. Train the model
trainer.fit(model, datamodule=datamodule)

# 5. Save the model!
trainer.save_checkpoint("image_classification_model.pt")
```

1.6 A few Built-in Tasks

- *ImageClassification*
- *ImageEmbedder*
- *TextClassification*
- *SummarizationTask*
- *TranslationTask*
- *TabularClassification*

More tasks coming soon!

1.6.1 Contribute a task

The lightning + Flash team is hard at work building more tasks for common deep-learning use cases. But we're looking for incredible contributors like you to submit new tasks!

Join our Slack to get help becoming a contributor!

CHAPTER

TWO

INSTALLATION

Flash is tested on Python 3.6+, and PyTorch 1.6.

2.1 Install with pip

```
pip install lightning-flash
```

Optionally, you can install Flash with extra packages for each domain or all domains.

```
pip install 'lightning-flash[image]'
pip install 'lightning-flash[tabular]'
pip install 'lightning-flash[text]'
pip install 'lightning-flash[video]'

# image + video
pip install 'lightning-flash[vision]'

# all features
pip install 'lightning-flash[all]'
```

For contributors, please install Flash with packages for testing Flash and building docs.

```
# Clone Flash repository locally
git clone https://github.com/[your username]/lightning-flash.git
cd lightning-flash

# Install Flash in editable mode with extra packages for development
pip install -e '.[dev]'
```

2.2 Install from source

```
pip install git+https://github.com/PyTorchLightning/lightning-flash.git
```

CHAPTER

THREE

TUTORIAL: CREATING A CUSTOM TASK

In this tutorial we will go over the process of creating a custom *Task*, along with a custom *DataModule*.

Note: This tutorial is only intended to help you create a small custom task for a personal project. If you want a more detailed guide, have a look at our *guide on contributing a task to flash.*

The tutorial objective is to create a `RegressionTask` to learn to predict if someone has `diabetes` or not. We will use `scikit-learn` Diabetes dataset. which is stored as numpy arrays.

Note: Find the complete tutorial example at flash_examples/custom_task.py.

3.1 1. Imports

We first import everything we're going to use and set the random seed using `seed_everything()`.

```
from typing import Any, Callable, Dict, List, Optional, Tuple

import numpy as np
import torch
from pytorch_lightning import seed_everything
from sklearn import datasets
from torch import nn, Tensor

import flash
from flash.core.data.data_source import DataSource, DefaultDataKeys, DefaultDataSources
from flash.core.data.process import Preprocess
from flash.core.data.transforms import ApplyToKeys

# set the random seeds.
seed_everything(42)

ND = np.ndarray
```

3.2 2. The Task: Linear regression

Here we create a basic linear regression task by subclassing *Task*. For the majority of tasks, you will likely need to override the `__init__`, `forward`, and the `{train,val,test,predict}_step` methods. The `__init__` should be overridden to configure the model and any additional arguments to be passed to the base *Task*. `forward` may need to be overridden to apply the model forward pass to the inputs. It's best practice in flash for the data to be provide as a dictionary which maps string keys to their values. The `{train,val,test,predict}_step` methods need to be overridden to extract the data from the input dictionary.

```
class RegressionTask(flash.Task):
    def __init__(self, num_inputs, learning_rate=0.2, metrics=None):
        # what kind of model do we want?
        model = torch.nn.Linear(num_inputs, 1)

        # what loss function do we want?
        loss_fn = torch.nn.functional.mse_loss

        # what optimizer to do we want?
        optimizer = torch.optim.Adam

        super().__init__(
            model=model,
            loss_fn=loss_fn,
            optimizer=optimizer,
            metrics=metrics,
            learning_rate=learning_rate,
        )

    def training_step(self, batch: Any, batch_idx: int) -> Any:
        return super().training_step(
            (batch[DefaultDataKeys.INPUT], batch[DefaultDataKeys.TARGET]),
            batch_idx,
        )

    def validation_step(self, batch: Any, batch_idx: int) -> None:
        return super().validation_step(
            (batch[DefaultDataKeys.INPUT], batch[DefaultDataKeys.TARGET]),
            batch_idx,
        )

    def test_step(self, batch: Any, batch_idx: int) -> None:
        return super().test_step(
            (batch[DefaultDataKeys.INPUT], batch[DefaultDataKeys.TARGET]),
            batch_idx,
        )

    def predict_step(self, batch: Any, batch_idx: int, dataloader_idx: int = 0) -> Any:
        return super().predict_step(
            batch[DefaultDataKeys.INPUT],
            batch_idx,
            dataloader_idx,
        )
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        # we don't actually need to override this method for this example
        return self.model(x)
```

Note: Lightning Flash provides registries. Registries are Flash internal key-value database to store a mapping between a name and a function. In simple words, they are just advanced dictionary storing a function from a key string. They are useful to store list of backbones and make them available for a *Task*. Check out *Available Registries* to learn more.

3.2.1 Where is the training step?

Most models can be trained simply by passing the output of `forward` to the supplied `loss_fn`, and then passing the resulting loss to the supplied `optimizer`. If you need a more custom configuration, you can override `step` (which is called for training, validation, and testing) or override `training_step`, `validation_step`, and `test_step` individually. These methods behave identically to PyTorch Lightning's methods.

Here is the pseudo code behind *Task* step:

```
def step(self, batch: Any, batch_idx: int) -> Any:
    """
    The training/validation/test step. Override for custom behavior.
    """
    x, y = batch
    y_hat = self(x)
    # compute the logs, loss and metrics as an output dictionary
    ...
    return output
```

3.3 3.a The DataSource API

Now that we have defined our `RegressionTask`, we need to load our data. We will define a custom `NumpyDataSource` which extends *DataSource*. The `NumpyDataSource` contains a `load_data` and `predict_load_data` methods which handle the loading of a sequence of dictionaries from the input numpy arrays. When loading the train data (`if self.training:`), the `NumpyDataSource` sets the `num_inputs` attribute of the optional `dataset` argument. Any attributes that are set on the optional `dataset` argument will also be set on the generated `dataset`.

```
class NumpyDataSource(DataSource[Tuple[ND, ND]]):
    def load_data(self, data: Tuple[ND, ND], dataset: Optional[Any] = None) ->␣
↪List[Dict[str, Any]]:
        if self.training:
            dataset.num_inputs = data[0].shape[1]
        return [{DefaultDataKeys.INPUT: x, DefaultDataKeys.TARGET: y} for x, y in␣
↪zip(*data)]

    def predict_load_data(self, data: ND) -> List[Dict[str, Any]]:
        return [{DefaultDataKeys.INPUT: x} for x in data]
```

3.4 3.b The Preprocess API

Now that we have a *DataSource* implementation, we can define our *Preprocess*. The *Preprocess* object provides a series of hooks that can be overridden with custom data processing logic and to which transforms can be attached. It allows the user much more granular control over their data processing flow.

Note: Why introduce *Preprocess* ?

The *Preprocess* object reduces the engineering overhead to make inference on raw data or to deploy the model in production environnement compared to a traditional Dataset.

You can override `predict_{hook_name}` hooks or the `default_predict_transforms` to handle data processing logic specific for inference.

The recommended way to define a custom *Preprocess* is as follows:

- Define an `__init__` which accepts transform arguments.
- **Pass these arguments through to `super().__init__` and specify the `data_sources` and the `default_data_source`.**
  - `data_sources` gives the *DataSource* objects that work with your *Preprocess* as a mapping from data source name to *DataSource*. The data source name can be any string, but for our purposes we can use `NUMPY` from *DefaultDataSources*.
  - `default_data_source` is the name of the data source to use by default when predicting.
- Override the `get_state_dict` and `load_state_dict` methods. These methods are used to save and load your *Preprocess* from a checkpoint.
- **Override the `{train,val,test,predict}_default_transforms` methods to specify the default transforms to use in eac**
  - Transforms are given as a mapping from hook name to callable transforms. You should use *ApplyToKeys* to apply each transform only to specific keys in the data dictionary.

```
class NumpyPreprocess(Preprocess):
    def __init__(
        self,
        train_transform: Optional[Dict[str, Callable]] = None,
        val_transform: Optional[Dict[str, Callable]] = None,
        test_transform: Optional[Dict[str, Callable]] = None,
        predict_transform: Optional[Dict[str, Callable]] = None,
    ):
        super().__init__(
            train_transform=train_transform,
            val_transform=val_transform,
            test_transform=test_transform,
            predict_transform=predict_transform,
            data_sources={DefaultDataSources.NUMPY: NumpyDataSource()},
            default_data_source=DefaultDataSources.NUMPY,
        )

    @staticmethod
    def to_float(x: Tensor):
        return x.float()
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def format_targets(x: Tensor):
        return x.unsqueeze(0)

    @property
    def to_tensor(self) -> Dict[str, Callable]:
        return {
            "to_tensor_transform": nn.Sequential(
                ApplyToKeys(
                    DefaultDataKeys.INPUT,
                    torch.from_numpy,
                    self.to_float,
                ),
                ApplyToKeys(
                    DefaultDataKeys.TARGET,
                    torch.as_tensor,
                    self.to_float,
                    self.format_targets,
                ),
            ),
        }

    def default_transforms(self) -> Optional[Dict[str, Callable]]:
        return self.to_tensor

    def get_state_dict(self) -> Dict[str, Any]:
        return self.transforms

    @classmethod
    def load_state_dict(cls, state_dict: Dict[str, Any], strict: bool = False):
        return cls(*state_dict)
```

3.5 3.c The DataModule API

Now that we have a *Preprocess* which knows about the *DataSource* objects it supports, we just need to create a *DataModule* which has a reference to the `preprocess_cls` we want it to use. For any data source whose name is in *DefaultDataSources*, there is a standard `DataModule.from_*` method that provides the expected inputs. So in this case, there is the *from_numpy()* that will use our numpy data source.

```
class NumpyDataModule(flash.DataModule):

    preprocess_cls = NumpyPreprocess
```

You now have a new customized Flash Task! Congratulations !

You can fit, finetune, validate and predict directly with those objects.

3.6 4. Fitting

For this task, here is how to fit the `RegressionTask` Task on `scikit-learn` Diabetes dataset.

Like any Flash Task, we can fit our model using the `flash.Trainer` by supplying the task itself, and the associated data:

```
x, y = datasets.load_diabetes(return_X_y=True)
datamodule = NumpyDataModule.from_numpy(x, y)

model = RegressionTask(num_inputs=datamodule.train_dataset.num_inputs)

trainer = flash.Trainer(
    max_epochs=20, progress_bar_refresh_rate=20, checkpoint_callback=False, gpus=torch.
↪cuda.device_count()
)
trainer.fit(model, datamodule=datamodule)
```

3.7 5. Predicting

With a trained model we can now perform inference. Here we will use a few examples from the test set of our data:

```
predict_data = np.array(
    [
        [0.0199, 0.0507, 0.1048, 0.0701, -0.0360, -0.0267, -0.0250, -0.0026, 0.0037, 0.
↪0403],
        [-0.0128, -0.0446, 0.0606, 0.0529, 0.0480, 0.0294, -0.0176, 0.0343, 0.0702, 0.
↪0072],
        [0.0381, 0.0507, 0.0089, 0.0425, -0.0428, -0.0210, -0.0397, -0.0026, -0.0181, 0.
↪0072],
        [-0.0128, -0.0446, -0.0235, -0.0401, -0.0167, 0.0046, -0.0176, -0.0026, -0.0385,␣
↪-0.0384],
        [-0.0237, -0.0446, 0.0455, 0.0907, -0.0181, -0.0354, 0.0707, -0.0395, -0.0345, -
↪0.0094],
    ]
)

predictions = model.predict(predict_data)
print(predictions)
```

We get the following output:

```
[tensor([189.1198]), tensor([196.0839]), tensor([161.2461]), tensor([130.7591]),␣
↪tensor([149.1780])]
```

CHAPTER

FOUR

FROM FLASH TO LIGHTNING

Flash is built on top of PyTorch Lightning to abstract away the unnecessary boilerplate for:

- Data science
- Kaggle
- Business use cases
- Applied research

Flash is a HIGH level library and Lightning is a LOW level library.

- Flash (high-level)
- Lightning (medium-level)
- PyTorch (low-level)

As the complexity increases or decreases, users can move between Flash and Lightning seamlessly to find the level of abstraction that works for them.

Table 1: Abstraction levels

Approach	Flexibility	Minimum DL Expertise level	PyTorch Knowledge	Use cases
Using an out-of-the-box task	Low	Novice+	Low+	Fast baseline, Data Science, Analysis, Applied Research
Using the Generic Task	Medium	Intermediate+	Intermediate+	Fast baseline, data science
Building a custom task	High	Intermediate+	Intermediate+	Fast baseline, custom business context, applied research
Building a LightningModule	Ultimate (organized PyTorch)	Expert+	Expert+	For anything you can do with PyTorch, AI research (academic and corporate)

4.1 Using an out-of-the-box task

Tasks can come from a variety of places:

- Flash
- Other Lightning-based libraries
- Your own library

Using a task requires almost zero knowledge of deep learning and PyTorch. The focus is on solving a problem as quickly as possible. This is great for:

- data science
- analysis
- applied research

4.2 Using the Generic Task

If you encounter a problem that does not have a matching task, you can use the generic task. However, this does require a bit of PyTorch knowledge but not a lot of knowledge over all the details of deep learning.

This is great for:

- data science
- kaggle baselines
- a quick baseline
- applied research
- learning about deep learning

Note: If you've used something like Keras, this is the most similar level of abstraction.

4.3 Building a custom task

If you're feeling adventurous and there isn't an out-of-the-box task for a particular applied problem, consider building your own task. This requires a decent amount of PyTorch knowledge, but not too much because tasks are LightningModules that already abstract a lot of the details for you.

This is great for:

- data science
- researchers building for corporate data science teams
- applied research
- custom business context

Note: In a company setting, a good setup here is to have your own Flash-like library with tasks contextualized with your business problems.

4.4 Building a LightningModule

Once you've reached the threshold of flexibility offered by Flash, it's time to move to a LightningModule directly. LightningModule is organized PyTorch but gives you the same flexibility. However, you must already know PyTorch fairly well and be comfortable with at least basic deep learning concepts.

This is great for:

- experts
- academic AI research
- corporate AI research
- advanced applied research
- publishing papers

CHAPTER

FIVE

FROM FLASH TO PRODUCTION

Flash makes it simple to deploy models in production.

5.1 Server Side

```
from flash.image import SemanticSegmentation
from flash.image.segmentation.serialization import SegmentationLabels

model = SemanticSegmentation.load_from_checkpoint(
    "https://flash-weights.s3.amazonaws.com/semantic_segmentation_model.pt"
)
model.serializer = SegmentationLabels(visualize=False)
model.serve()
```

5.2 Client Side

```
import base64
from pathlib import Path

import requests

import flash

with (Path(flash.ASSETS_ROOT) / "road.png").open("rb") as f:
    imgstr = base64.b64encode(f.read()).decode("UTF-8")

body = {"session": "UUID", "payload": {"inputs": {"data": imgstr}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)
print(resp.json())
```

Credits to @rlizzo, @hhsecond, @lantiga, @luiscape for building Flash Serve Engine.

CHAPTER

SIX

TRAINING FROM SCRATCH

Some Flash tasks have been pretrained on large data sets. To accelerate your training, calling the *finetune()* method using a pretrained backbone will fine-tune the backbone to generate a model customized to your data set and desired task.

From the *Quick Start* guide.

To train a task from scratch:

1. Load your data and organize it using a DataModule customized for the task (example: `ImageClassificationData`).
2. Choose and initialize your Task (setting `pretrained=False`) which has state-of-the-art backbones built in (example: `ImageClassifier`).
3. Init a *flash.core.trainer.Trainer* or a `pytorch_lightning.trainer.Trainer`.
4. Call *flash.core.trainer.Trainer.fit()* with your data set.
5. Save your trained model.

Here's an example:

```
from pytorch_lightning import seed_everything

import flash
from flash.core.classification import Labels
from flash.core.data.utils import download_data
from flash.image import ImageClassificationData, ImageClassifier

# set the random seeds.
seed_everything(42)

# 1. Download and organize the data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
)
```

(continues on next page)

(continued from previous page)

```
# 2. Build the model using desired Task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes,␣
↪pretrained=False)

# 3. Create the trainer (run one epoch for demo)
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())

# 4. Train the model
trainer.fit(model, datamodule=datamodule)

# 5. Save the model!
trainer.save_checkpoint("image_classification_model.pt")
```

6.1 Training options

Flash tasks supports many advanced training functionalities out-of-the-box, such as:

- limit number of epochs

```
# train for 10 epochs
flash.Trainer(max_epochs=10)
```

- Training on GPUs

```
# train on 1 GPU
flash.Trainer(gpus=1)
```

- Training on multiple GPUs

```
# train on multiple GPUs
flash.Trainer(gpus=4)
```

```
# train on gpu 1, 3, 5 (3 gpus total)
flash.Trainer(gpus=[1, 3, 5])
```

- Using mixed precision training

```
# Multi GPU with mixed precision
flash.Trainer(gpus=2, precision=16)
```

- Training on TPUs

```
# Train on TPUs
flash.Trainer(tpu_cores=8)
```

You can add to the flash Trainer any argument from the Lightning trainer! Learn more about the Lightning Trainer here.

CHAPTER

SEVEN

FINETUNING

Finetuning (or transfer-learning) is the process of tweaking a model trained on a large dataset, to your particular (likely much smaller) dataset.

7.1 Terminology

Here are common terms you need to be familiar with:

Table 1: Terminology

Term	Definition
Finetuning	The process of tweaking a model trained on a large dataset, to your particular (likely much smaller) dataset
Transfer learning	The common name for finetuning
Backbone	The neural network that was pretrained on a different dataset
Head	Another neural network (usually smaller) that maps the backbone to your particular dataset
Freeze	Disabling gradient updates to a model (ie: not learning)
Unfreeze	Enabling gradient updates to a model

7.2 Finetuning in Flash

From the *Quick Start* guide.

To use a Task for finetuning:

1. Load your data and organize it using a DataModule customized for the task (example: `ImageClassificationData`).
2. Choose and initialize your Task which has state-of-the-art backbones built in (example: `ImageClassifier`).
3. Init a *flash.core.trainer.Trainer*.
4. Choose a finetune strategy (example: "freeze") and call *flash.core.trainer.Trainer.finetune()* with your data.
5. Save your finetuned model.

Here's an example of finetuning.

```
from pytorch_lightning import seed_everything

import flash
from flash.core.classification import Labels
from flash.core.data.utils import download_data
from flash.image import ImageClassificationData, ImageClassifier

# set the random seeds.
seed_everything(42)

# 1. Download and organize the data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
)

# 2. Build the model using desired Task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes)

# 3. Create the trainer (run one epoch for demo)
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())

# 4. Finetune the model
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 5. Save the model!
trainer.save_checkpoint("image_classification_model.pt")
```

7.2.1 Using a finetuned model

Once you've finetuned, use the model to predict:

```
# Serialize predictions as labels, automatically inferred from the training data in part␣
↪2.
model.serializer = Labels()

predictions = model.predict(
    [
        "data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg",
        "data/hymenoptera_data/val/ants/2255445811_dabcdf7258.jpg",
    ]
)
print(predictions)
```

We get the following output:

```
['bees', 'ants']
```

Or you can use the saved model for prediction anywhere you want!

```
from flash.image import ImageClassifier

# load finetuned checkpoint
model = ImageClassifier.load_from_checkpoint("image_classification_model.pt")

predictions = model.predict("path/to/your/own/image.png")
```

7.3 Finetune strategies

Finetuning is very task specific. Each task encodes the best finetuning practices for that task. However, Flash gives you a few default strategies for finetuning.

Finetuning operates on two things, the model backbone and the head. The backbone is the neural network that was pre-trained. The head is another neural network that bridges between the backbone and your particular dataset.

7.3.1 no_freeze

In this strategy, the backbone and the head are unfrozen from the beginning.

```
trainer.finetune(model, datamodule, strategy="no_freeze")
```

In pseudocode, this looks like:

```
backbone = Resnet50()
head = nn.Linear(...)

backbone.unfreeze()
head.unfreeze()

train(backbone, head)
```

7.3.2 freeze

The freeze strategy keeps the backbone frozen throughout.

```
trainer.finetune(model, datamodule, strategy="freeze")
```

The pseudocode looks like:

```
backbone = Resnet50()
head = nn.Linear(...)

# freeze backbone
backbone.freeze()
head.unfreeze()

train(backbone, head)
```

7.4 Advanced strategies

Every finetune strategy can also be customized.

7.4.1 freeze_unfreeze

By default, in this strategy the backbone is frozen for 5 epochs then unfrozen:

```
trainer.finetune(model, datamodule, strategy="freeze_unfreeze")
```

Or we can customize it unfreeze the backbone after a different epoch. For example, to unfreeze after epoch 7:

```
from flash.core.finetuning import FreezeUnfreeze

trainer.finetune(model, datamodule, strategy=FreezeUnfreeze(unfreeze_epoch=7))
```

Under the hood, the pseudocode looks like:

```
backbone = Resnet50()
head = nn.Linear(...)

# freeze backbone
backbone.freeze()
head.unfreeze()

train(backbone, head, epochs=10)

# unfreeze after 10 epochs
backbone.unfreeze()

train(backbone, head)
```

7.4.2 unfreeze_milestones

This strategy allows you to unfreeze part of the backbone at predetermined intervals

Here's an example where: - backbone starts frozen - at epoch 3 the last 2 layers unfreeze - at epoch 8 the full backbone unfreezes

```
from flash.core.finetuning import UnfreezeMilestones

trainer.finetune(model, datamodule, strategy=UnfreezeMilestones(unfreeze_milestones=(3,␣
↪8), num_layers=2))
```

Under the hood, the pseudocode looks like:

```
backbone = Resnet50()
head = nn.Linear(...)

# freeze backbone
backbone.freeze()
head.unfreeze()

train(backbone, head, epochs=3)

# unfreeze last 2 layers at epoch 3
backbone.unfreeze_last_layers(2)

train(backbone, head, epochs=8)

# unfreeze the full backbone
backbone.unfreeze()
```

7.5 Custom Strategy

For even more customization, create your own finetuning callback. Learn more about callbacks here.

```
from flash.core.finetuning import FlashBaseFinetuning

# Create a finetuning callback
class FeatureExtractorFreezeUnfreeze(FlashBaseFinetuning):
    def __init__(self, unfreeze_epoch: int = 5, train_bn: bool = True):
        # this will set self.attr_names as ["backbone"]
        super().__init__("backbone", train_bn)
        self._unfreeze_epoch = unfreeze_epoch

    def finetune_function(self, pl_module, current_epoch, optimizer, opt_idx):
        # unfreeze any module you want by overriding this function

        # When ``current_epoch`` is 5, backbone will start to be trained.
        if current_epoch == self._unfreeze_epoch:
            self.unfreeze_and_add_param_group(
                pl_module.backbone,
                optimizer,
            )


# Pass the callback to trainer.finetune
trainer.finetune(model, datamodule, strategy=FeatureExtractorFreezeUnfreeze(unfreeze_
↪epoch=5))
```

CHAPTER

EIGHT

PREDICTIONS (INFERENCE)

You can use Flash to get predictions on pretrained or finetuned models.

8.1 Predict on a single sample of data

You can pass in a sample of data (image file path, a string of text, etc) to the *predict()* method.

```
from flash.core.data.utils import download_data
from flash.image import ImageClassifier


# 1. Download the data set
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

# 2. Load the model from a checkpoint
model = ImageClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪image_classification_model.pt")

# 3. Predict whether the image contains an ant or a bee
predictions = model.predict("data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg")
print(predictions)
```

8.2 Predict on a csv file

```
from flash.core.data.utils import download_data
from flash.tabular import TabularClassifier

# 1. Download the data
download_data("https://pl-flash-data.s3.amazonaws.com/titanic.zip", "data/")

# 2. Load the model from a checkpoint
model = TabularClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪tabnet_classification_model.pt")

# 3. Generate predictions from a csv file! Who would survive?
predictions = model.predict("data/titanic/titanic.csv")
print(predictions)
```

8.3 Serializing predictions

To change how predictions are serialized you can attach a *Serializer* to your *Task*. For example, you can choose to serialize outputs as probabilities (for more options see the API reference below).

```
from flash.core.classification import Probabilities
from flash.core.data.utils import download_data
from flash.image import ImageClassifier


# 1. Download the data set
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

# 2. Load the model from a checkpoint
model = ImageClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪image_classification_model.pt")

# 3. Attach the Serializer
model.serializer = Probabilities()

# 4. Predict whether the image contains an ant or a bee
predictions = model.predict("data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg")
print(predictions)
# out: [[0.5926494598388672, 0.407350480556488804]]
```

CHAPTER

NINE

TORCHSCRIPT JIT SUPPORT

We test all of our tasks for compatibility with `torch.jit`. This table gives a breakdown of the supported features.

Task	`torch.jit.script()`	`torch.jit.trace()`	`torch.jit.save()`
ImageClassifier	Yes	Yes	Yes
ObjectDetector	Yes	No	Yes
ImageEmbedder	Yes	Yes	Yes
SemanticSegmentation	No	Yes	Yes
StyleTransfer	No	Yes	Yes
TabularClassifier	No	Yes	No
TextClassifier	No	Yes *	Yes
SummarizationTask	No	Yes	Yes
TranslationTask	No	Yes	Yes
VideoClassifier	No	Yes	Yes

* with `strict=False`

CHAPTER

TEN

DATA

10.1 Terminology

Here are common terms you need to be familiar with:

Table 1: Terminology

Term	Definition
Deserializer	The *Deserializer* provides a single `deserialize()` method.
DataModule	The *DataModule* contains the datasets, transforms and dataloaders.
DataPipeline	The *DataPipeline* is Flash internal object to manage `Deserializer`, *DataSource*, *Preprocess*, *Postprocess*, and *Serializer* objects.
DataSource	The *DataSource* provides *load_data()* and *load_sample()* hooks for creating data sets from metadata (such as folder names).
Preprocess	**The *Preprocess* provides a simple hook-based API to encapsulate your pre-processing logic.** These hooks (such as *pre_tensor_transform()*) enable transformations to be applied to your data at every point along the pipeline (including on the device). The *DataPipeline* contains a system to call the right hooks when needed. The *Preprocess* hooks can be either overridden directly or provided as a dictionary of transforms (mapping hook name to callable transform).
Postprocess	**The *Postprocess* provides a simple hook-based API to encapsulate your post-processing logic.** The *Postprocess* hooks cover from model outputs to predictions export.
Serializer	The *Serializer* provides a single *serialize()* method that is used to convert model outputs (after the *Postprocess*) to the desired output format during prediction.

10.2 How to use out-of-the-box Flash DataModules

Flash provides several DataModules with helpers functions. Check out the *Image Classification* section (or the sections for any of our other tasks) to learn more.

10.3 Data Processing

Currently, it is common practice to implement a `torch.utils.data.Dataset` and provide it to a `torch.utils.data.DataLoader`. However, after model training, it requires a lot of engineering overhead to make inference on raw data and deploy the model in production environment. Usually, extra processing logic should be added to bridge the gap between training data and raw data.

The *DataSource* class can be used to generate data sets from multiple sources (e.g. folders, numpy, etc.), that can then all be transformed in the same way. The *Preprocess* and *Postprocess* classes can be used to manage the preprocessing and postprocessing transforms. The *Serializer* class provides the logic for converting *Postprocess* outputs to the desired predict format (e.g. classes, labels, probabilities, etc.).

By providing a series of hooks that can be overridden with custom data processing logic (or just targeted with transforms), Flash gives the user much more granular control over their data processing flow.

Here are the primary advantages:

- Making inference on raw data simple
- Make the code more readable, modular and self-contained
- Data Augmentation experimentation is simpler

To change the processing behavior only on specific stages for a given hook, you can prefix each of the *Preprocess* and *Postprocess* hooks by adding `train`, `val`, `test` or `predict`.

Check out *Preprocess* for some examples.

10.4 How to customize existing DataModules

Any Flash *DataModule* can be created directly from datasets using the *from_datasets()* like this:

```
from flash import DataModule, Trainer

data_module = DataModule.from_datasets(train_dataset=MyDataset())
trainer = Trainer()
trainer.fit(model, data_module=data_module)
```

The *DataModule* provides additional `classmethod` helpers (`from_*`) for loading data from various sources. In each `from_*` method, the *DataModule* internally retrieves the correct *DataSource* to use from the *Preprocess*. Flash *AutoDataset* instances are created from the *DataSource* for train, val, test, and predict. The *DataModule* populates the `DataLoader` for each stage with the corresponding *AutoDataset*.

10.5 Customize preprocessing of DataModules

The *Preprocess* contains the processing logic related to a given task. Each *Preprocess* provides some default transforms through the *default_transforms()* method. Users can easily override these by providing their own transforms to the *DataModule*. Here's an example:

```
from flash.core.data.transforms import ApplyToKeys
from flash.image import ImageClassificationData, ImageClassifier

transform = {"to_tensor_transform": ApplyToKeys("input", my_to_tensor_transform)}

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
    train_transform=transform,
    val_transform=transform,
    test_transform=transform,
)
```

Alternatively, the user may directly override the hooks for their needs like this:

```
from typing import Any, Dict
from flash.image import ImageClassificationData, ImageClassifier,␣
↪ImageClassificationPreprocess


class CustomImageClassificationPreprocess(ImageClassificationPreprocess):
    def to_tensor_transform(sample: Dict[str, Any]) -> Dict[str, Any]:
        sample["input"] = my_to_tensor_transform(sample["input"])
        return sample


datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
    preprocess=CustomImageClassificationPreprocess(),
)
```

10.6 Create your own Preprocess and DataModule

The example below shows a very simple `ImageClassificationPreprocess` with a single `ImageClassificationFoldersDataSource` and an `ImageClassificationDataModule`.

10.6.1 1. User-Facing API design

Designing an easy-to-use API is key. This is the first and most important step. We want the `ImageClassificationDataModule` to generate a dataset from folders of images arranged in this way.

Example:

```
train/dog/xxx.png
train/dog/xxy.png
train/dog/xxz.png
train/cat/123.png
train/cat/nsdf3.png
train/cat/asd932.png
```

Example:

```
dm = ImageClassificationDataModule.from_folders(
    train_folder="./data/train",
    val_folder="./data/val",
    test_folder="./data/test",
    predict_folder="./data/predict",
)

model = ImageClassifier(...)
trainer = Trainer(...)

trainer.fit(model, dm)
```

10.6.2 2. The DataSource

We start by implementing the `ImageClassificationFoldersDataSource`. The `load_data` method will produce a list of files and targets from the given directory. The `load_sample` method will load the given file as a `PIL.Image`. Here's the full `ImageClassificationFoldersDataSource`:

```
from PIL import Image
from torchvision.datasets.folder import make_dataset
from typing import Any, Dict
from flash.core.data.data_source import DataSource, DefaultDataKeys


class ImageClassificationFoldersDataSource(DataSource):
    def load_data(self, folder: str, dataset: Any) -> Iterable:
        # The dataset is optional but can be useful to save some metadata.

        # `metadata` contains the image path and its corresponding label
        # with the following structure:
```

(continues on next page)

(continued from previous page)

```
        # [(image_path_1, label_1), ... (image_path_n, label_n)].
        metadata = make_dataset(folder)

        # for the train `AutoDataset`, we want to store the `num_classes`.
        if self.training:
            dataset.num_classes = len(np.unique([m[1] for m in metadata]))

        return [
            {
                DefaultDataKeys.INPUT: file,
                DefaultDataKeys.TARGET: target,
            }
            for file, target in metadata
        ]

    def predict_load_data(self, predict_folder: str) -> Iterable:
        # This returns [image_path_1, ... image_path_m].
        return [{DefaultDataKeys.INPUT: file} for file in os.listdir(folder)]

    def load_sample(self, sample: Dict[str, Any]) -> Dict[str, Any]:
        sample[DefaultDataKeys.INPUT] = Image.open(sample[DefaultDataKeys.INPUT])
        return sample
```

Note: We return samples as dictionaries using the *DefaultDataKeys* by convention. This is the recommended (although not required) way to represent data in Flash.

10.6.3 3. The Preprocess

Next, implement your custom `ImageClassificationPreprocess` with some default transforms and a reference to the data source:

```
from typing import Any, Callable, Dict, Optional
from flash.core.data.data_source import DefaultDataKeys, DefaultDataSources
from flash.core.data.process import Preprocess
import torchvision.transforms.functional as T

# Subclass `Preprocess`
class ImageClassificationPreprocess(Preprocess):
    def __init__(
        self,
        train_transform: Optional[Dict[str, Callable]] = None,
        val_transform: Optional[Dict[str, Callable]] = None,
        test_transform: Optional[Dict[str, Callable]] = None,
        predict_transform: Optional[Dict[str, Callable]] = None,
    ):
        super().__init__(
            train_transform=train_transform,
            val_transform=val_transform,
            test_transform=test_transform,
```

(continues on next page)

(continued from previous page)

```
            predict_transform=predict_transform,
            data_sources={
                DefaultDataSources.FOLDERS: ImageClassificationFoldersDataSource(),
            },
            default_data_source=DefaultDataSources.FOLDERS,
        )

    def get_state_dict(self) -> Dict[str, Any]:
        return {**self.transforms}

    @classmethod
    def load_state_dict(cls, state_dict: Dict[str, Any], strict: bool = False):
        return cls(**state_dict)

    def default_transforms(self) -> Dict[str, Callable]:
        return {"to_tensor_transform": ApplyToKeys(DefaultDataKeys.INPUT, T.to_tensor)}
```

10.6.4 4. The DataModule

Finally, let's implement the `ImageClassificationDataModule`. We get the `from_folders` classmethod for free as we've registered a `DefaultDataSources.FOLDERS` data source in our `ImageClassificationPreprocess`. All we need to do is attach our *Preprocess* class like this:

```
from flash import DataModule


class ImageClassificationDataModule(DataModule):

    # Set `preprocess_cls` with your custom `Preprocess`.
    preprocess_cls = ImageClassificationPreprocess
```

10.7 How it works behind the scenes

10.7.1 DataSource

Note: The *load_data()* and *load_sample()* will be used to generate an *AutoDataset* object.

Here is the *AutoDataset* pseudo-code.

```
class AutoDataset:
    def __init__(
        self,
        data: List[Any],  # output of `DataSource.load_data`
        data_source: DataSource,
        running_stage: RunningStage,
    ):
```

(continues on next page)

(continued from previous page)

```
        self.data = data
        self.data_source = data_source

    def __getitem__(self, index: int):
        return self.data_source.load_sample(self.data[index])

    def __len__(self):
        return len(self.data)
```

10.7.2 Preprocess

Note: The *pre_tensor_transform()*, *to_tensor_transform()*, *post_tensor_transform()*, *collate()*, *per_batch_transform()* are injected as the `torch.utils.data.DataLoader.collate_fn` function of the DataLoader.

Here is the pseudo code using the preprocess hooks name. Flash takes care of calling the right hooks for each stage.

Example:

```
# This will be wrapped into a :class:`~flash.core.data.batch._Preprocessor`.
def collate_fn(samples: Sequence[Any]) -> Any:

    # This will be wrapped into a :class:`~flash.core.data.batch._Sequential`
    for sample in samples:
        sample = pre_tensor_transform(sample)
        sample = to_tensor_transform(sample)
        sample = post_tensor_transform(sample)

    samples = type(samples)(samples)

    # if :func:`flash.core.data.process.Preprocess.per_sample_transform_on_device` hook␣
↪is overridden,
    # those functions below will be no-ops

    samples = collate(samples)
    samples = per_batch_transform(samples)
    return samples

dataloader = DataLoader(dataset, collate_fn=collate_fn)
```

Note: The `per_sample_transform_on_device`, `collate`, `per_batch_transform_on_device` are injected after the `LightningModule` `transfer_batch_to_device` hook.

Here is the pseudo code using the preprocess hooks name. Flash takes care of calling the right hooks for each stage.

Example:

```
# This will be wrapped into a :class:`~flash.core.data.batch._Preprocessor`
def collate_fn(samples: Sequence[Any]) -> Any:
```

(continues on next page)

(continued from previous page)

```
    # if ``per_batch_transform`` hook is overridden, those functions below will be no-ops
    samples = [per_sample_transform_on_device(sample) for sample in samples]
    samples = type(samples)(samples)
    samples = collate(samples)

    samples = per_batch_transform_on_device(samples)
    return samples

# move the data to device
data = lightning_module.transfer_data_to_device(data)
data = collate_fn(data)
predictions = lightning_module(data)
```

10.7.3 Postprocess and Serializer

Once the predictions have been generated by the Flash *Task*, the Flash *DataPipeline* will execute the *Postprocess* hooks and the *Serializer* behind the scenes.

First, the *per_batch_transform()* hooks will be applied on the batch predictions. Then, the *uncollate()* will split the batch into individual predictions. Next, the *per_sample_transform()* will be applied on each prediction. Finally, the *serialize()* method will be called to serialize the predictions.

Note: The transform can be applied either on device or CPU.

Here is the pseudo-code:

Example:

```
# This will be wrapped into a :class:`~flash.core.data.batch._Postprocessor`
def uncollate_fn(batch: Any) -> Any:

    batch = per_batch_transform(batch)

    samples = uncollate(batch)

    samples = [per_sample_transform(sample) for sample in samples]
    # only if serializers are enabled.
    return [serialize(sample) for sample in samples]

predictions = lightning_module(data)
return uncollate_fn(predictions)
```

CHAPTER

ELEVEN

REGISTRY

11.1 Available Registries

Registries are Flash internal key-value database to store a mapping between a name and a function.

In simple words, they are just advanced dictionary storing a function from a key string.

Registries help organize code and make the functions accessible all across the `Flash` codebase. Each Flash *Task* can have several registries as static attributes.

Currently, Flash uses internally registries only for backbones, but more components will be added.

11.1.1 1. Imports

```
from functools import partial

from flash import Task
from flash.core.registry import FlashRegistry
```

11.1.2 2. Init a Registry

It is good practice to associate one or multiple registry to a Task as follow:

```
# creating a custom `Task` with its own registry
class MyImageClassifier(Task):

    backbones = FlashRegistry("backbones")

    def __init__(
        self,
        backbone: str = "resnet18",
        pretrained: bool = True,
    ):
        ...

        self.backbone, self.num_features = self.backbones.
↪get(backbone)(pretrained=pretrained)
```

11.1.3 3. Adding new functions

Your custom functions can be registered within a *FlashRegistry* as a decorator or directly.

```
# Option 1: Used with partial.
def fn(backbone: str, pretrained: bool = True):
    # Create backbone and backbone output dimension (`num_features`)
    backbone, num_features = None, None
    return backbone, num_features


# HINT 1: Use `from functools import partial` if you want to store some arguments.
MyImageClassifier.backbones(fn=partial(fn, backbone="my_backbone"), name="username/
↪partial_backbone")

# Option 2: Using decorator.
@MyImageClassifier.backbones(name="username/decorated_backbone")
def fn(pretrained: bool = True):
    # Create backbone and backbone output dimension (`num_features`)
    backbone, num_features = None, None
    return backbone, num_features
```

11.1.4 4. Accessing registered functions

You can now access your function from your task!

```
# 3.b Optional: List available backbones
print(MyImageClassifier.available_backbones())

# 4. Build the model
model = MyImageClassifier(backbone="username/decorated_backbone")
```

Here's the output:

```
['username/decorated_backbone', 'username/partial_backbone']
```

11.1.5 5. Pre-registered backbones

Flash provides populated registries containing lots of available backbones.

Example:

```
from flash.image.backbones import OBJ_DETECTION_BACKBONES
from flash.image.classification.backbones import IMAGE_CLASSIFIER_BACKBONES

print(IMAGE_CLASSIFIER_BACKBONES.available_keys())
""" out:
['adv_inception_v3', 'cspdarknet53', 'cspdarknet53_iabn', 430+.., 'xception71']
"""
```

CHAPTER

TWELVE

FLASH ZERO

Flash Zero is a zero-code machine learning platform built directly into lightning-flash. To get started and view the available tasks, run:

```
flash --help
```

12.1 Customize Trainer and Model arguments

Flash Zero is built on top of the lightning CLI, so the trainer and model arguments can be configured either from the command line or from a config file. For example, to run the image classifier for 10 epochs with a *resnet50* backbone you can use:

```
flash image_classification --trainer.max_epochs 10 --model.backbone resnet50
```

To view all of the available options for a task, run:

```
flash image_classification --help
```

12.2 Using Custom Data

Flash Zero works with your own data through subcommands. The available subcommands for each task are given at the bottom of their help pages (e.g. when running `flash image-classification --help`). You can then use the required subcommand to train on your own data. Let's look at an example using the Hymenoptera data from the *Image Classification* guide. First, download and unzip your data:

```
curl https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip -o hymenoptera_data
unzip hymenoptera_data.zip
```

Now train with Flash Zero:

```
flash image_classification from_folders --train_folder ./hymenoptera_data/train
```

You can view the help page for each subcommand. For example, to view the options for training an image classifier from folders, you can run:

```
flash image_classification from_folders --help
```

CHAPTER

THIRTEEN

FLASH SERVE

Flash Serve is a library to easily serve models in production.

13.1 Terminology

Here are common terms you need to be familiar with:

Table 1: Terminology

Term	Definition
de-serialization	Transform data encoded as text into tensors
inference function	A function taking the decoded tensors and forward them through the model to produce predictions.
serialization	Transform the predictions tensors back to a text encoding.
`ModelComponent`	The `ModelComponent` contains the de-serialization, inference and serialization functions.
`Servable`	The `Servable` is an helper track the asset file related to a model
`Composition`	The `Composition` defines the computations / endpoints to create & run
`expose()`	The *`expose()`* function is a python decorator used to augment the `ModelComponent` inference function with de-serialization, serialization.

13.2 Example

In this tutorial, we will serve a Resnet18 from the PyTorchVision library in 3 steps.

The entire tutorial can be found under `flash_examples/serve/generic`.

13.2.1 Introduction

Traditionally, an inference pipeline is made out of 3 steps:

- `de-serialization`: Transform data encoded as text into tensors.
- `inference function`: A function taking the decoded tensors and forward them through the model to produce predictions.
- `serialization`: Transform the predictions tensors back as text.

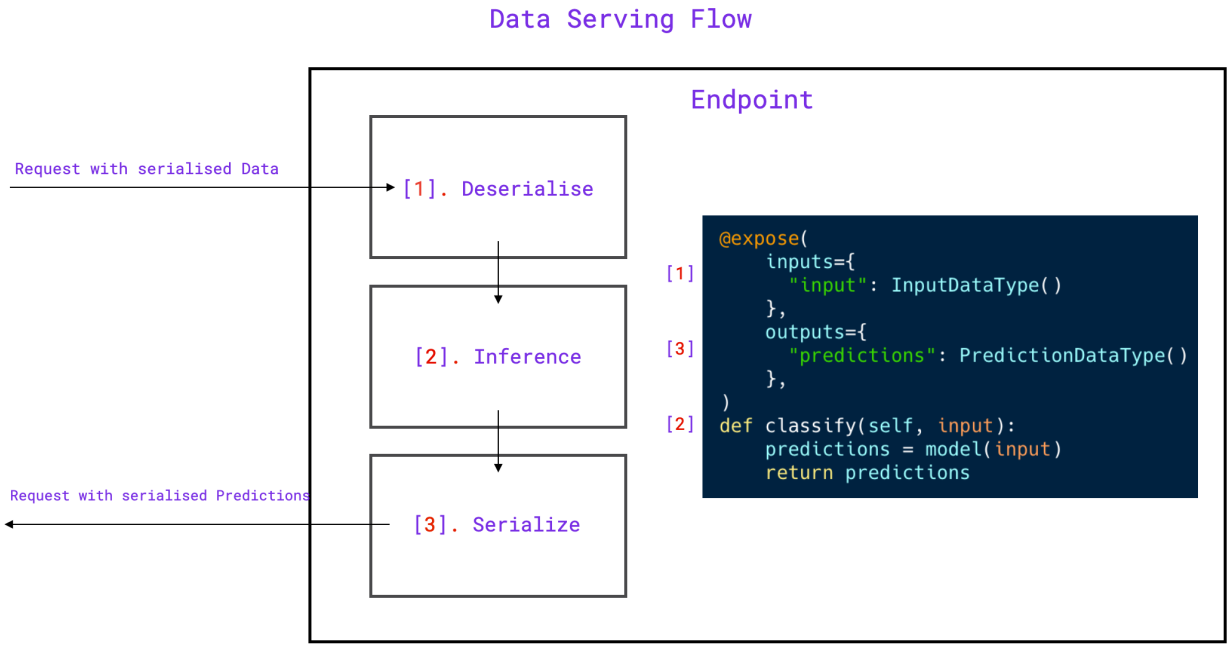
In this example, we will implement only the inference function as Flash Serve already provides some built-in `de-serialization` and `serialization` functions with `Image`

13.2.2 Step 1 - Create a ModelComponent

Inside `inference_serve.py`, we will implement a `ClassificationInference` class, which overrides `ModelComponent`.

First, we need make the following imports:

```
import torch
import torchvision

from flash.core.serve import Composition, Servable, ModelComponent, expose
from flash.core.serve.types import Image, Label
```



To implement `ClassificationInference`, we need to implement a method responsible for `inference function` and decorated with the *`expose()`* function.

The name of the inference method isn't constrained, but we will use `classify` as appropriate in this example.

Our classify function will take a tensor image, apply some normalization on it, and forward it through the model.

```
def classify(img):
    img = img.float() / 255
    mean = torch.tensor([[[0.485, 0.456, 0.406]]]).float()
    std = torch.tensor([[[0.229, 0.224, 0.225]]]).float()
    img = (img - mean) / std
    img = img.permute(0, 3, 2, 1)
    out = self.model(img)
    return out.argmax()
```

The *`expose()`* is a python decorator extending the decorated function with the `de-serialization`, `serialization` steps.

Note: Flash Serve was designed this way to enable several models to be chained together by removing the decorator.

The *expose()* function takes 2 arguments:

- `inputs`: Dictionary mapping the decorated function inputs to `BaseType` objects.
- `outputs`: Dictionary mapping the decorated function outputs to `BaseType` objects.

A `BaseType` is a python dataclass which implements a `serialize` and `deserialize` function.

Note: Flash Serve has already several `BaseType` built-in such as `Image` or `Text`.

```
class ClassificationInference(ModelComponent):
    def __init__(self, model: Servable):
        self.model = model

    @expose(
        inputs={"img": Image()},
        outputs={"prediction": Label(path="imagenet_labels.txt")},
    )
    def classify(self, img):
        img = img.float() / 255
        mean = torch.tensor([[[0.485, 0.456, 0.406]]]).float()
        std = torch.tensor([[[0.229, 0.224, 0.225]]]).float()
        img = (img - mean) / std
        img = img.permute(0, 3, 2, 1)
        out = self.model(img)
        return out.argmax()
```

13.2.3 Step 2 - Create a scripted Model

Using the PyTorchVision library, we create a `resnet18` and use torch.jit.script to script the model.

Note: TorchScript is a way to create serializable and optimizable models from PyTorch code. Any TorchScript program can be saved from a Python process and loaded in a process where there is no Python dependency.

```
model = torchvision.models.resnet18(pretrained=True).eval()
torch.jit.script(model).save("resnet.pt")
```

13.2.4 Step 3 - Serve the model

The `Servable` takes as argument the path to the TorchScripted model and then will be passed to our `ClassificationInference` class.

The `ClassificationInference` instance will be passed as argument to a `Composition` class.

Once the `Composition` class is instantiated, just call its `serve()` method.

```
resnet = Servable("resnet.pt")
comp = ClassificationInference(resnet)
composition = Composition(classification=comp)
composition.serve()
```

13.2.5 Launching the server.

In Terminal 1

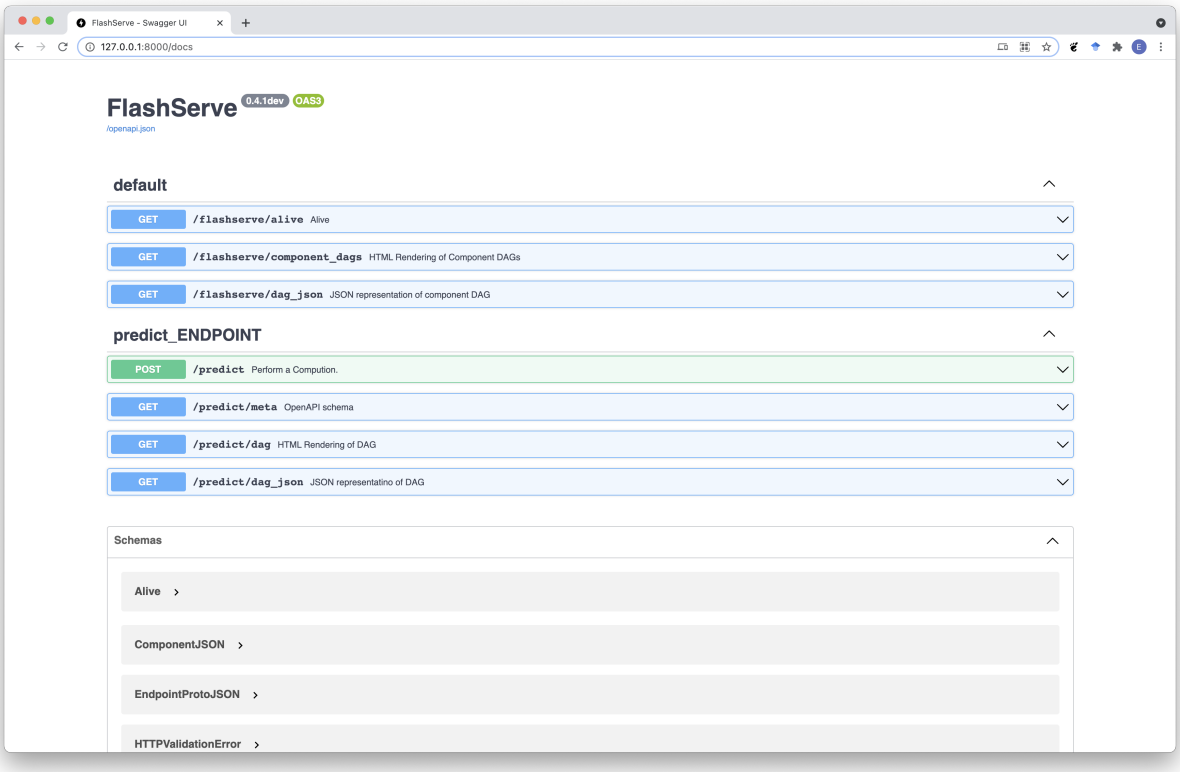
Just run:

```
python inference_server.py
```

And you should see this in your terminal

```
(.venv) ➜  image_classification git:(master) ✗ python inference_server.py
INFO:     Started server process [636]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

You should also see an Swagger UI already built for you at `http://127.0.0.1:8000/docs`



In Terminal 2

Run this script from another terminal:

```
import base64
from pathlib import Path

import requests

with Path("fish.jpg").open("rb") as f:
    imgstr = base64.b64encode(f.read()).decode("UTF-8")

body = {"session": "UUID", "payload": {"img": {"data": imgstr}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)
print(resp.json())
# {'session': 'UUID', 'result': {'prediction': 'goldfish, Carassius auratus'}}
```

Credits to @rlizzo, @hhsecond, @lantiga, @luiscape for building Flash Serve Engine.

CHAPTER

FOURTEEN

IMAGE CLASSIFICATION

14.1 The Task

The task of identifying what is in an image is called image classification. Typically, Image Classification is used to identify images containing a single object. The task predicts which 'class' the image most likely belongs to with a degree of certainty. A class is a label that describes what is in an image, such as 'car', 'house', 'cat' etc.

14.2 Example

Let's look at the task of predicting whether images contain Ants or Bees using the hymenoptera dataset. The dataset contains `train` and `validation` folders, and then each folder contains a **bees** folder, with pictures of bees, and an **ants** folder with images of, you guessed it, ants.

```
hymenoptera_data
├── train
│   ├── ants
│   │   ├── 0013035.jpg
│   │   ├── 1030023514_aad5c608f9.jpg
│   │   ...
│   └── bees
│       ├── 1092977343_cb42b38d62.jpg
│       ├── 1093831624_fb5fbe2308.jpg
│       ...
└── val
    ├── ants
    │   ├── 10308379_1b6c72e180.jpg
    │   ├── 1053149811_f62a3410d3.jpg
    │   ...
    └── bees
        ├── 1032546534_06907fe3b3.jpg
        ├── 10870992_eebeeb3a12.jpg
        ...
```

Once we've downloaded the data using `download_data()`, we create the *ImageClassificationData*. We select a pre-trained backbone to use for our *ImageClassifier* and fine-tune on the hymenoptera data. We then use the trained *ImageClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.image import ImageClassificationData, ImageClassifier

# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "./data")

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
)

# 2. Build the task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Predict what's on a few images! ants or bees?
predictions = model.predict(
    [
        "data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg",
        "data/hymenoptera_data/val/bees/590318879_68cf112861.jpg",
        "data/hymenoptera_data/val/ants/540543309_ddbb193ee5.jpg",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("image_classification_model.pt")
```

14.3 Flash Zero

The image classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the hymenoptera example with:

```
flash image_classification
```

To view configuration options and options for running the image classifier with your own data, use:

```
flash image_classification --help
```

14.4 Loading Data

This section details the available ways to load your own data into the *ImageClassificationData*.

14.4.1 from_folders

Construct the *ImageClassificationData* from folders.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp, .npy.

For train, test, and val data, the folders are expected to contain a sub-folder for each class. Here's the required structure:

```
train_folder
├── class_1
│   ├── file1.jpg
│   ├── file2.jpg
│   ...
└── class_2
    ├── file1.jpg
    ├── file2.jpg
    ...
```

For prediction, the folder is expected to contain the files for inference, like this:

```
predict_folder
├── file1.jpg
├── file2.jpg
...
```

Example:

```
data_module = ImageClassificationData.from_folders(
    train_folder = "./train_folder",
    predict_folder = "./predict_folder",
    ...
)
```

14.4.2 from_files

Construct the *ImageClassificationData* from lists of files and corresponding lists of targets.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp, .npy.

Example:

```
train_files = ["file1.jpg", "file2.jpg", "file3.jpg", ...]
train_targets = [0, 1, 0, ...]

datamodule = ImageClassificationData.from_files(
    train_files = train_files,
    train_targets = train_targets,
    ...
)
```

14.4.3 from_datasets

Construct the *ImageClassificationData* from the given datasets for each stage.

Example:

```
from torch.utils.data.dataset import Dataset

train_dataset: Dataset = ...

datamodule = ImageClassificationData.from_datasets(
    train_dataset = train_dataset,
    ...
)
```

Note: The `__getitem__` of your datasets should return a dictionary with `"input"` and `"target"` keys which map to the input image (as a PIL.Image) and the target (as an int or list of ints) respectively.

14.5 Custom Transformations

Flash automatically applies some default image transformations and augmentations, but you may wish to customize these for your own use case. The base *Preprocess* defines 7 hooks for different stages in the data loading pipeline. To apply image augmentations you can directly import the `default_transforms` from `flash.image.classification.transforms` and then merge your custom image transformations with them using the *merge_transforms()* helper function. Here's an example where we load the default transforms and merge with custom *torchvision* transformations. We use the *post_tensor_transform* hook to apply the transformations after the image has been converted to a *torch.Tensor*.

```
from torchvision import transforms as T

import flash
from flash.core.data.data_source import DefaultDataKeys
from flash.core.data.transforms import ApplyToKeys, merge_transforms
from flash.image import ImageClassificationData, ImageClassifier
from flash.image.classification.transforms import default_transforms

post_tensor_transform = ApplyToKeys(
    DefaultDataKeys.INPUT,
    T.Compose([T.RandomHorizontalFlip(), T.ColorJitter(), T.RandomAutocontrast(), T.
↪RandomPerspective()]),
)

new_transforms = merge_transforms(default_transforms((64, 64)), {"post_tensor_transform
↪": post_tensor_transform})

datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/", val_folder="data/hymenoptera_data/val/",
↪ train_transform=new_transforms
)
```

(continues on next page)

(continued from previous page)

```
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes)

trainer = flash.Trainer(max_epochs=1)
trainer.finetune(model, datamodule=datamodule, strategy="freeze")
```

14.6 Serving

The *ImageClassifier* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.image import ImageClassifier

model = ImageClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪image_classification_model.pt")
model.serve()
```

You can now perform inference from your client like this:

```
import base64
from pathlib import Path

import requests

import flash

with (Path(flash.ASSETS_ROOT) / "fish.jpg").open("rb") as f:
    imgstr = base64.b64encode(f.read()).decode("UTF-8")

body = {"session": "UUID", "payload": {"inputs": {"data": imgstr}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)
print(resp.json())
```

CHAPTER

FIFTEEN

MULTI-LABEL IMAGE CLASSIFICATION

15.1 The Task

Multi-label classification is the task of assigning a number of labels from a fixed set to each data point, which can be in any modality (images in this case). Multi-label image classification is supported by the *ImageClassifier* via the `multi-label` argument.

15.2 Example

Let's look at the task of trying to predict the movie genres from an image of the movie poster. The data we will use is a subset of the awesome movie poster genre prediction data set from the paper "Movie Genre Classification based on Poster Images with Deep Neural Networks" by Wei-Ta Chu and Hung-Jui Guo, resized to 128 by 128. Take a look at their paper (and please consider citing their paper if you use the data) here: www.cs.ccu.edu.tw/~wtchu/projects/MoviePoster/. The data set contains `train` and `validation` folders, and then each folder contains images and a `metadata.csv` which stores the labels. Here's an overview:

```
movie_posters
├── train
│   ├── metadata.csv
│   ├── tt0084058.jpg
│   ├── tt0084867.jpg
│   ...
└── val
    ├── metadata.csv
    ├── tt0200465.jpg
    ├── tt0326965.jpg
    ...
```

Once we've downloaded the data using `download_data()`, we need to create the *ImageClassificationData*. We first create a function (`load_data`) to extract the list of images and associated labels which can then be passed to `from_files()`. We select a pre-trained backbone to use for our *ImageClassifier* and fine-tune on the posters data. We then use the trained *ImageClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
```

(continues on next page)

(continued from previous page)

```
from flash.image import ImageClassificationData, ImageClassifier

# 1. Create the DataModule
# Data set from the paper “Movie Genre Classification based on Poster Images with Deep␣
↪Neural Networks”.
# More info here: https://www.cs.ccu.edu.tw/~wtchu/projects/MoviePoster/
download_data("https://pl-flash-data.s3.amazonaws.com/movie_posters.zip")

datamodule = ImageClassificationData.from_csv(
    "Id",
    ["Action", "Romance", "Crime", "Thriller", "Adventure"],
    train_file="data/movie_posters/train/metadata.csv",
    val_file="data/movie_posters/val/metadata.csv",
    image_size=(128, 128),
)

# 2. Build the task
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes, multi_
↪label=True)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Predict the genre of a few movies!
predictions = model.predict(
    [
        "data/movie_posters/predict/tt0085318.jpg",
        "data/movie_posters/predict/tt0089461.jpg",
        "data/movie_posters/predict/tt0097179.jpg",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("image_classification_multi_label_model.pt")
```

15.3 Flash Zero

The multi-label image classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the movie posters example with:

```
flash image_classification from_movie_posters
```

To view configuration options and options for running the image classifier with your own data, use:

```
flash image_classification --help
```

15.4 Serving

The `ImageClassifier` is servable. For more information, see *Image Classification*.

CHAPTER

SIXTEEN

IMAGE EMBEDDER

16.1 The Task

Image embedding encodes an image into a vector of features which can be used for a downstream task. This could include: clustering, similarity search, or classification.

16.2 Example

Let's see how to use the *ImageEmbedder* with a pretrained backbone to obtain feature vectors from the hymenoptera data. Once we've downloaded the data, we create the *ImageEmbedder* and perform inference (obtaining feature vectors / embeddings) using `predict()`. Here's the full example:

```
from flash.core.data.utils import download_data
from flash.image import ImageEmbedder

# 1. Download the data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip", "data/")

# 2. Build the task
embedder = ImageEmbedder(backbone="resnet101")

# 3. Generate an embedding from an image path.
embeddings = embedder.predict(["data/hymenoptera_data/predict/153783656_85f9c3ac70.jpg"])
print(embeddings)
```

CHAPTER

SEVENTEEN

OBJECT DETECTION

17.1 The Task

Object detection is the task of identifying objects in images and their associated classes and bounding boxes.

The *ObjectDetector* and *ObjectDetectionData* classes internally rely on IceVision.

17.2 Example

Let's look at object detection with the COCO 128 data set, which contains 91 object classes. This is a subset of COCO train2017 with only 128 images. The data set is organized following the COCO format. Here's an outline:

```
coco128
├── annotations
│   └── instances_train2017.json
├── images
│   └── train2017
│       ├── 000000000009.jpg
│       ├── 000000000025.jpg
│       ...
└── labels
    └── train2017
        ├── 000000000009.txt
        ├── 000000000025.txt
        ...
```

Once we've downloaded the data using `download_data()`, we can create the *ObjectDetectionData*. We select a pre-trained RetinaNet to use for our *ObjectDetector* and fine-tune on the COCO 128 data. We then use the trained *ObjectDetector* for inference. Finally, we save the model. Here's the full example:

```
import flash
from flash.core.data.utils import download_data
from flash.image import ObjectDetectionData, ObjectDetector

# 1. Create the DataModule
# Dataset Credit: https://www.kaggle.com/ultralytics/coco128
download_data("https://github.com/zhiqwang/yolov5-rt-stack/releases/download/v0.3.0/
↪coco128.zip", "data/")
```

(continues on next page)

(continued from previous page)

```
datamodule = ObjectDetectionData.from_coco(
    train_folder="data/coco128/images/train2017/",
    train_ann_file="data/coco128/annotations/instances_train2017.json",
    val_split=0.1,
    image_size=128,
)

# 2. Build the task
model = ObjectDetector(head="efficientdet", backbone="d0", num_classes=datamodule.num_
↪classes, image_size=128)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=1)
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Detect objects in a few images!
predictions = model.predict(
    [
        "data/coco128/images/train2017/000000000625.jpg",
        "data/coco128/images/train2017/000000000626.jpg",
        "data/coco128/images/train2017/000000000629.jpg",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("object_detection_model.pt")
```

17.3 Flash Zero

The object detector can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash object_detection
```

To view configuration options and options for running the object detector with your own data, use:

```
flash object_detection --help
```

CHAPTER

EIGHTEEN

KEYPOINT DETECTION

18.1 The Task

Keypoint detection is the task of identifying keypoints in images and their associated classes.

The *KeypointDetector* and *KeypointDetectionData* classes internally rely on IceVision.

18.2 Example

Let's look at keypoint detection with BIWI Sample Keypoints (center of face) from IceData. Once we've downloaded the data, we can create the *KeypointDetectionData*. We select a `keypoint_rcnn` with a `resnet18_fpn` backbone to use for our *KeypointDetector* and fine-tune on the BIWI data. We then use the trained *KeypointDetector* for inference. Finally, we save the model. Here's the full example:

```
import flash
from flash.core.utilities.imports import example_requires
from flash.image import KeypointDetectionData, KeypointDetector

example_requires("image")

import icedata  # noqa: E402

# 1. Create the DataModule
data_dir = icedata.biwi.load_data()

datamodule = KeypointDetectionData.from_folders(
    train_folder=data_dir,
    val_split=0.1,
    parser=icedata.biwi.parser,
)

# 2. Build the task
model = KeypointDetector(
    head="keypoint_rcnn",
    backbone="resnet18_fpn",
    num_keypoints=1,
    num_classes=datamodule.num_classes,
)
```

(continues on next page)

(continued from previous page)

```
# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=1)
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Detect objects in a few images!
predictions = model.predict(
    [
        str(data_dir / "biwi_sample/images/0.jpg"),
        str(data_dir / "biwi_sample/images/1.jpg"),
        str(data_dir / "biwi_sample/images/10.jpg"),
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("keypoint_detection_model.pt")
```

18.3 Flash Zero

The keypoint detector can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash keypoint_detection
```

To view configuration options and options for running the keypoint detector with your own data, use:

```
flash keypoint_detection --help
```

CHAPTER

NINETEEN

INSTANCE SEGMENTATION

19.1 The Task

Instance segmentation is the task of segmenting objects images and determining their associated classes.

The *InstanceSegmentation* and *InstanceSegmentationData* classes internally rely on IceVision.

19.2 Example

Let's look at instance segmentation with The Oxford-IIIT Pet Dataset from IceData. Once we've downloaded the data, we can create the *InstanceSegmentationData*. We select a `mask_rcnn` with a `resnet18_fpn` backbone to use for our *InstanceSegmentation* and fine-tune on the pets data. We then use the trained *InstanceSegmentation* for inference. Finally, we save the model. Here's the full example:

```
from functools import partial

import flash
from flash.core.utilities.imports import example_requires
from flash.image import InstanceSegmentation, InstanceSegmentationData

example_requires("image")

import icedata  # noqa: E402

# 1. Create the DataModule
data_dir = icedata.pets.load_data()

datamodule = InstanceSegmentationData.from_folders(
    train_folder=data_dir,
    val_split=0.1,
    parser=partial(icedata.pets.parser, mask=True),
)

# 2. Build the task
model = InstanceSegmentation(
    head="mask_rcnn",
    backbone="resnet18_fpn",
    num_classes=datamodule.num_classes,
```

(continues on next page)

(continued from previous page)

```
)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=1)
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Detect objects in a few images!
predictions = model.predict(
    [
        str(data_dir / "images/yorkshire_terrier_9.jpg"),
        str(data_dir / "images/english_cocker_spaniel_1.jpg"),
        str(data_dir / "images/scottish_terrier_1.jpg"),
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("instance_segmentation_model.pt")
```

19.3 Flash Zero

The instance segmentation task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash instance_segmentation
```

To view configuration options and options for running the instance segmentation task with your own data, use:

```
flash instance_segmentation --help
```

CHAPTER

TWENTY

SEMANTIC SEGMENTATION

20.1 The Task

Semantic Segmentation, or image segmentation, is the task of performing classification at a pixel-level, meaning each pixel will associated to a given class. See more: https://paperswithcode.com/task/semantic-segmentation

20.2 Example

Let's look at an example using a data set generated with the CARLA driving simulator. The data was generated as part of the Kaggle Lyft Udacity Challenge. The data contains one folder of images and another folder with the corresponding segmentation masks. Here's the structure:

```
data
├── CameraRGB
│   ├── F61-1.png
│   ├── F61-2.png
│       ...
└── CameraSeg
    ├── F61-1.png
    ├── F61-2.png
        ...
```

Once we've downloaded the data using `download_data()`, we create the *SemanticSegmentationData*. We select a pre-trained `mobilenet_v3_large` backbone with an `fpn` head to use for our *SemanticSegmentation* task and fine-tune on the CARLA data. We then use the trained *SemanticSegmentation* for inference. You can check the available pretrained weights for the backbones like this *SemanticSegmentation.available_pretrained_weights("resnet18")*. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.image import SemanticSegmentation, SemanticSegmentationData

# 1. Create the DataModule
# The data was generated with the  CARLA self-driving simulator as part of the Kaggle␣
↪Lyft Udacity Challenge.
# More info here: https://www.kaggle.com/kumaresanmanickavelu/lyft-udacity-challenge
```

(continues on next page)

(continued from previous page)

```
download_data(
    "https://github.com/ongchinkiat/LyftPerceptionChallenge/releases/download/v0.1/carla-
↪capture-20180513A.zip",
    "./data",
)

datamodule = SemanticSegmentationData.from_folders(
    train_folder="data/CameraRGB",
    train_target_folder="data/CameraSeg",
    val_split=0.1,
    image_size=(256, 256),
    num_classes=21,
)

# 2. Build the task
model = SemanticSegmentation(
    backbone="mobilenetv3_large_100",
    head="fpn",
    num_classes=datamodule.num_classes,
)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Segment a few images!
predictions = model.predict(
    [
        "data/CameraRGB/F61-1.png",
        "data/CameraRGB/F62-1.png",
        "data/CameraRGB/F63-1.png",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("semantic_segmentation_model.pt")
```

20.3 Flash Zero

The semantic segmentation task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash semantic_segmentation
```

To view configuration options and options for running the semantic segmentation task with your own data, use:

```
flash semantic_segmentation --help
```

20.4 Loading Data

This section details the available ways to load your own data into the *SemanticSegmentationData*.

20.4.1 from_folders

Construct the *SemanticSegmentationData* from folders.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp.

For train, test, and val data, we expect a folder containing inputs and another folder containing the masks. Here's the required structure:

```
train_folder
├── inputs
    ├── file1.jpg
    ├── file2.jpg
    ...
└── masks
    ├── file1.jpg
    ├── file2.jpg
    ...
```

For prediction, the folder is expected to contain the files for inference, like this:

```
predict_folder
├── file1.jpg
├── file2.jpg
...
```

Example:

```
data_module = SemanticSegmentationData.from_folders(
    train_folder = "./train_folder/inputs",
    train_target_folder = "./train_folder/masks",
    predict_folder = "./predict_folder",
    ...
)
```

20.4.2 from_files

Construct the *SemanticSegmentationData* from lists of input images and corresponding list of target images.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp.

Example:

```
train_files = ["file1.jpg", "file2.jpg", "file3.jpg", ...]
train_targets = ["mask1.jpg", "mask2.jpg", "mask3.jpg", ...]

datamodule = SemanticSegmentationData.from_files(
    train_files = train_files,
    train_targets = train_targets,
```

(continues on next page)

(continued from previous page)

```
    ...
)
```

20.4.3 from_datasets

Construct the *SemanticSegmentationData* from the given datasets for each stage.

Example:

```
from torch.utils.data.dataset import Dataset

train_dataset: Dataset = ...

datamodule = SemanticSegmentationData.from_datasets(
    train_dataset = train_dataset,
    ...
)
```

Note: The `__getitem__` of your datasets should return a dictionary with `"input"` and `"target"` keys which map to the input and target images as tensors.

20.5 Serving

The *SemanticSegmentation* task is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.image import SemanticSegmentation
from flash.image.segmentation.serialization import SegmentationLabels

model = SemanticSegmentation.load_from_checkpoint(
    "https://flash-weights.s3.amazonaws.com/semantic_segmentation_model.pt"
)
model.serializer = SegmentationLabels(visualize=False)
model.serve()
```

You can now perform inference from your client like this:

```
import base64
from pathlib import Path

import requests

import flash

with (Path(flash.ASSETS_ROOT) / "road.png").open("rb") as f:
    imgstr = base64.b64encode(f.read()).decode("UTF-8")
```

(continues on next page)

(continued from previous page)

```
body = {"session": "UUID", "payload": {"inputs": {"data": imgstr}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)
print(resp.json())
```

CHAPTER

TWENTYONE

STYLE TRANSFER

21.1 The Task

The Neural Style Transfer Task is an optimization method which extract the style from an image and apply it another image while preserving its content. The goal is that the output image looks like the content image, but "painted" in the style of the style reference image.

The *StyleTransfer* and *StyleTransferData* classes internally rely on pystiche.

21.2 Example

Let's look at transferring the style from The Starry Night onto the images from the COCO 128 data set from the *Object Detection* Guide. Once we've downloaded the data using `download_data()`, we create the *StyleTransferData*. Next, we create our *StyleTransfer* task with the desired style image and fit on the COCO 128 images. We then use the trained *StyleTransfer* for inference. Finally, we save the model. Here's the full example:

```
import os

import torch

import flash
from flash.core.data.utils import download_data
from flash.image.style_transfer import StyleTransfer, StyleTransferData

# 1. Create the DataModule
download_data("https://github.com/zhiqwang/yolov5-rt-stack/releases/download/v0.3.0/
↪coco128.zip", "./data")

datamodule = StyleTransferData.from_folders(train_folder="data/coco128/images/train2017")
```

(continues on next page)

(continued from previous page)

```
# 2. Build the task
model = StyleTransfer(os.path.join(flash.ASSETS_ROOT, "starry_night.jpg"))

# 3. Create the trainer and train the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.fit(model, datamodule=datamodule)

# 4. Apply style transfer to a few images!
predictions = model.predict(
    [
        "data/coco128/images/train2017/000000000625.jpg",
        "data/coco128/images/train2017/000000000626.jpg",
        "data/coco128/images/train2017/000000000629.jpg",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("style_transfer_model.pt")
```

21.3 Flash Zero

The style transfer task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash style_transfer
```

To view configuration options and options for running the style transfer task with your own data, use:

```
flash style_transfer --help
```

CHAPTER

TWENTYTWO

VIDEO CLASSIFICATION

22.1 The Task

Typically, Video Classification refers to the task of producing a label for actions identified in a given video. The task is to predict which *class* the video clip belongs to.

Lightning Flash `VideoClassifier` and `VideoClassificationData` classes internally rely on PyTorchVideo.

22.2 Example

Let's develop a model to classifying video clips of Humans performing actions (such as: **archery** , **bowling**, etc.). We'll use data from the Kinetics dataset. Here's an outline of the folder structure:

```
video_dataset
├── train
│   ├── archery
│   │   ├── -1q7jA3DXQM_000005_000015.mp4
│   │   ├── -5NN5hdIwTc_000036_000046.mp4
│   │   ...
│   ├── bowling
│   │   ├── -5ExwuF5IUI_000030_000040.mp4
│   │   ├── -7sTNNI1Bcg_000075_000085.mp4
│   ... ...
└── val
    ├── archery
    │   ├── 0S-P4lr_c7s_000022_000032.mp4
    │   ├── 2x1lIrgKxYo_000589_000599.mp4
    │   ...
    ├── bowling
    │   ├── 1W7HNDBA4pA_000002_000012.mp4
    │   ├── 4JxH3S5JwMs_000003_000013.mp4
    ... ...
```

Once we've downloaded the data using `download_data()`, we create the `VideoClassificationData`. We select a pre-trained backbone to use for our `VideoClassifier` and fine-tune on the Kinetics data. The backbone can be any model from the PyTorchVideo Model Zoo. We then use the trained `VideoClassifier` for inference. Finally, we save the model. Here's the full example:

```
import os

import torch

import flash
from flash.core.data.utils import download_data
from flash.video import VideoClassificationData, VideoClassifier

# 1. Create the DataModule
# Find more datasets at https://pytorchvideo.readthedocs.io/en/latest/data.html
download_data("https://pl-flash-data.s3.amazonaws.com/kinetics.zip", "./data")

datamodule = VideoClassificationData.from_folders(
    train_folder=os.path.join(os.getcwd(), "data/kinetics/train"),
    val_folder=os.path.join(os.getcwd(), "data/kinetics/val"),
    clip_sampler="uniform",
    clip_duration=1,
    decode_audio=False,
)

# 2. Build the task
model = VideoClassifier(backbone="x3d_xs", num_classes=datamodule.num_classes,␣
↪pretrained=False)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Make a prediction
predictions = model.predict(os.path.join(os.getcwd(), "data/kinetics/predict"))
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("video_classification.pt")
```

22.3 Flash Zero

The video classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash video_classification
```

To view configuration options and options for running the video classifier with your own data, use:

```
flash video_classification --help
```

CHAPTER

TWENTYTHREE

AUDIO CLASSIFICATION

23.1 The Task

The task of identifying what is in an audio file is called audio classification. Typically, Audio Classification is used to identify audio files containing sounds or words. The task predicts which 'class' the sound or words most likely belongs to with a degree of certainty. A class is a label that describes the sounds in an audio file, such as 'children_playing', 'jackhammer', 'siren' etc.

23.2 Example

Let's look at the task of predicting whether audio file contains sounds of an airconditioner, carhorn, childrenplaying, dogbark, drilling, engingeidling, gunshot, jackhammer, siren, or street_music using the UrbanSound8k spectrogram images dataset. The dataset contains `train`, `val` and `test` folders, and then each folder contains a **airconditioner** folder, with spectrograms generated from air-conditioner sounds, **siren** folder with spectrograms generated from siren sounds and the same goes for the other classes.

```
urban8k_images
├── train
│   ├── air_conditioner
│   ├── car_horn
│   ├── children_playing
│   ├── dog_bark
│   ├── drilling
│   ├── engine_idling
│   ├── gun_shot
│   ├── jackhammer
│   ├── siren
│   └── street_music
├── test
│   ├── air_conditioner
│   ├── car_horn
│   ├── children_playing
│   ├── dog_bark
│   ├── drilling
│   ├── engine_idling
│   ├── gun_shot
│   ├── jackhammer
```

(continues on next page)

(continued from previous page)

```
│   ├── siren
│   └── street_music
└── val
    ├── air_conditioner
    ├── car_horn
    ├── children_playing
    ├── dog_bark
    ├── drilling
    ├── engine_idling
    ├── gun_shot
    ├── jackhammer
    ├── siren
    └── street_music

        ...
```

Once we've downloaded the data using `download_data()`, we create the *AudioClassificationData*. We select a pre-trained backbone to use for our *ImageClassifier* and fine-tune on the UrbanSound8k spectrogram images data. We then use the trained *ImageClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.audio import AudioClassificationData
from flash.core.data.utils import download_data
from flash.core.finetuning import FreezeUnfreeze
from flash.image import ImageClassifier

# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/urban8k_images.zip", "./data")

datamodule = AudioClassificationData.from_folders(
    train_folder="data/urban8k_images/train",
    val_folder="data/urban8k_images/val",
    spectrogram_size=(64, 64),
)

# 2. Build the model.
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy=FreezeUnfreeze(unfreeze_epoch=1))

# 4. Predict what's on few images! air_conditioner, children_playing, siren e.t.c
predictions = model.predict(
    [
        "data/urban8k_images/test/air_conditioner/13230-0-0-5.wav.jpg",
        "data/urban8k_images/test/children_playing/9223-2-0-15.wav.jpg",
        "data/urban8k_images/test/jackhammer/22883-7-10-0.wav.jpg",
    ]
)
```

(continues on next page)

(continued from previous page)

```
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("audio_classification_model.pt")
```

23.3 Flash Zero

The audio classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash audio_classification
```

To view configuration options and options for running the audio classifier with your own data, use:

```
flash audio_classification --help
```

23.4 Loading Data

This section details the available ways to load your own data into the *AudioClassificationData*.

23.4.1 from_folders

Construct the *AudioClassificationData* from folders.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp, .npy.

For train, test, and val data, the folders are expected to contain a sub-folder for each class. Here's the required structure:

```
train_folder
├── class_1
│   ├── file1.jpg
│   ├── file2.jpg
│   ...
└── class_2
    ├── file1.jpg
    ├── file2.jpg
    ...
```

For prediction, the folder is expected to contain the files for inference, like this:

```
predict_folder
├── file1.jpg
├── file2.jpg
...
```

Example:

```
data_module = AudioClassificationData.from_folders(
    train_folder = "./train_folder",
    predict_folder = "./predict_folder",
    ...
)
```

23.4.2 from_files

Construct the *AudioClassificationData* from lists of files and corresponding lists of targets.

The supported file extensions are: .jpg, .jpeg, .png, .ppm, .bmp, .pgm, .tif, .tiff, .webp, .npy.

Example:

```
train_files = ["file1.jpg", "file2.jpg", "file3.jpg", ...]
train_targets = [0, 1, 0, ...]

datamodule = AudioClassificationData.from_files(
    train_files = train_files,
    train_targets = train_targets,
    ...
)
```

23.4.3 from_datasets

Construct the *AudioClassificationData* from the given datasets for each stage.

Example:

```
from torch.utils.data.dataset import Dataset

train_dataset: Dataset = ...

datamodule = AudioClassificationData.from_datasets(
    train_dataset = train_dataset,
    ...
)
```

Note: The `__getitem__` of your datasets should return a dictionary with `"input"` and `"target"` keys which map to the input spectrogram image (as a NumPy array) and the target (as an int or list of ints) respectively.

CHAPTER

TWENTYFOUR

SPEECH RECOGNITION

24.1 The Task

Speech recognition is the task of classifying audio into a text transcription. We rely on Wav2Vec as our backbone, fine-tuned on labeled transcriptions for speech to text. Wav2Vec is pre-trained on thousand of hours of unlabeled audio, providing a strong baseline when fine-tuning to downstream tasks such as Speech Recognition.

24.2 Example

Let's fine-tune the model onto our own labeled audio transcription data:

Here's the structure our CSV file:

```
file,text
"/path/to/file_1.wav","what was said in file 1."
"/path/to/file_2.wav","what was said in file 2."
"/path/to/file_3.wav","what was said in file 3."
...
```

Alternatively, here is the structure of our JSON file:

```
{"file": "/path/to/file_1.wav", "text": "what was said in file 1."}
{"file": "/path/to/file_2.wav", "text": "what was said in file 2."}
{"file": "/path/to/file_3.wav", "text": "what was said in file 3."}
```

Once we've downloaded the data using `download_data()`, we create the *SpeechRecognitionData*. We select a pre-trained Wav2Vec backbone to use for our *SpeechRecognition* and finetune on a subset of the TIMIT corpus. The backbone can be any Wav2Vec model from HuggingFace transformers. Next, we use the trained *SpeechRecognition* for inference and save the model. Here's the full example:

```
import torch

import flash
from flash.audio import SpeechRecognition, SpeechRecognitionData
from flash.core.data.utils import download_data

# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/timit_data.zip", "./data")
```

(continues on next page)

(continued from previous page)

```
datamodule = SpeechRecognitionData.from_json(
    input_fields="file",
    target_fields="text",
    train_file="data/timit/train.json",
    test_file="data/timit/test.json",
)

# 2. Build the task
model = SpeechRecognition(backbone="facebook/wav2vec2-base-960h")

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="no_freeze")

# 4. Predict on audio files!
predictions = model.predict(["data/timit/example.wav"])
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("speech_recognition_model.pt")
```

24.3 Flash Zero

The speech recognition task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash speech_recognition
```

To view configuration options and options for running the speech recognition task with your own data, use:

```
flash speech_recognition --help
```

24.4 Serving

The *SpeechRecognition* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.audio import SpeechRecognition

model = SpeechRecognition.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪speech_recognition_model.pt")
model.serve()
```

You can now perform inference from your client like this:

```
import base64
from pathlib import Path

import requests

import flash

with (Path(flash.ASSETS_ROOT) / "example.wav").open("rb") as f:
    audio_str = base64.b64encode(f.read()).decode("UTF-8")

body = {"session": "UUID", "payload": {"inputs": {"data": audio_str}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)

print(resp.json())
```

CHAPTER

TWENTYFIVE

TABULAR CLASSIFICATION

25.1 The Task

Tabular classification is the task of assigning a class to samples of structured or relational data. The *TabularClassifier* task can be used for classification of samples in more than two classes (multi-class classification).

25.2 Example

Let's look at training a model to predict if passenger survival on the Titanic using the classic Kaggle data set. The data is provided in CSV files that look like this:

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
3,1,3,"Heikkinen, Miss. Laina",female,26,0,0,STON/O2. 3101282,7.925,,S
5,0,3,"Allen, Mr. William Henry",male,35,0,0,373450,8.05,,S
6,0,3,"Moran, Mr. James",male,,0,0,330877,8.4583,,Q
...
```

Once we've downloaded the data using `download_data()`, we can create the `TabularData` from our CSV files using the `from_csv()` method. From `the API reference`, we need to provide:

- **cat_cols**- A list of the names of columns that contain categorical data (strings or integers).
- **num_cols**- A list of the names of columns that contain numerical continuous data (floats).
- **target**- The name of the column we want to predict.
- **train_csv**- A CSV file containing the training data converted to a Pandas DataFrame

Next, we create the *TabularClassifier* and finetune on the Titanic data. We then use the trained *TabularClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.tabular import TabularClassificationData, TabularClassifier

# 1. Create the DataModule
```

(continues on next page)

(continued from previous page)

```
download_data("https://pl-flash-data.s3.amazonaws.com/titanic.zip", "./data")

datamodule = TabularClassificationData.from_csv(
    ["Sex", "Age", "SibSp", "Parch", "Ticket", "Cabin", "Embarked"],
    "Fare",
    target_fields="Survived",
    train_file="data/titanic/titanic.csv",
    val_split=0.1,
)

# 2. Build the task
model = TabularClassifier.from_data(datamodule)

# 3. Create the trainer and train the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.fit(model, datamodule=datamodule)

# 4. Generate predictions from a CSV
predictions = model.predict("data/titanic/titanic.csv")
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("tabular_classification_model.pt")
```

25.3 Flash Zero

The tabular classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash tabular_classifier
```

To view configuration options and options for running the tabular classifier with your own data, use:

```
flash tabular_classifier --help
```

25.4 Serving

The *TabularClassifier* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.core.classification import Labels
from flash.tabular import TabularClassifier

model = TabularClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪tabular_classification_model.pt")
model.serializer = Labels(["Did not survive", "Survived"])
model.serve()
```

You can now perform inference from your client like this:

```
import pandas as pd
import requests

from flash.core.data.utils import download_data

# 1. Download the data
download_data("https://pl-flash-data.s3.amazonaws.com/titanic.zip", "data/")

df = pd.read_csv("./data/titanic/predict.csv")
text = str(df.to_csv())
body = {"session": "UUID", "payload": {"inputs": {"data": text}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)
print(resp.json())
```

CHAPTER

TWENTYSIX

TEXT CLASSIFICATION

26.1 The Task

Text classification is the task of assigning a piece of text (word, sentence or document) an appropriate class, or category. The categories depend on the chosen data set and can range from topics.

26.2 Example

Let's train a model to classify text as expressing either positive or negative sentiment. We will be using the IMDB data set, that contains a `train.csv` and `valid.csv`. Here's the structure:

```
review,sentiment
"Japanese indie film with humor ... ",positive
"Isaac Florentine has made some ...",negative
"After seeing the low-budget ...",negative
"I've seen the original English version ...",positive
"Hunters chase what they think is a man through ...",negative
...
```

Once we've downloaded the data using `download_data()`, we create the *TextClassificationData*. We select a pre-trained backbone to use for our *TextClassifier* and finetune on the IMDB data. The backbone can be any BERT classification model from HuggingFace/transformers.

Note: When changing the backbone, make sure you pass in the same backbone to the *TextClassifier* and the *TextClassificationData*!

Next, we use the trained *TextClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.text import TextClassificationData, TextClassifier

# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/imdb.zip", "./data/")
```

(continues on next page)

(continued from previous page)

```
datamodule = TextClassificationData.from_csv(
    "review",
    "sentiment",
    train_file="data/imdb/train.csv",
    val_file="data/imdb/valid.csv",
    backbone="prajjwal1/bert-medium",
)

# 2. Build the task
model = TextClassifier(backbone="prajjwal1/bert-medium", num_classes=datamodule.num_
↪classes)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Classify a few sentences! How was the movie?
predictions = model.predict(
    [
        "Turgid dialogue, feeble characterization - Harvey Keitel a judge?.",
        "The worst movie in the history of cinema.",
        "I come from Bulgaria where it 's almost impossible to have a tornado.",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("text_classification_model.pt")
```

26.3 Flash Zero

The text classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash text_classification
```

To view configuration options and options for running the text classifier with your own data, use:

```
flash text_classification --help
```

26.4 Serving

The *TextClassifier* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.text import TextClassifier

model = TextClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/text_
↪classification_model.pt")
model.serve()
```

You can now perform inference from your client like this:

```
import requests

text = "Best movie ever"
body = {"session": "UUID", "payload": {"inputs": {"data": text}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)

print(resp.json())
```

26.5 Accelerate Training & Inference with Torch ORT

Torch ORT converts your model into an optimized ONNX graph, speeding up training & inference when using NVIDIA or AMD GPUs. Enabling Torch ORT requires a single flag passed to the `TextClassifier` once installed. See installation instructions here.

Note: Not all Transformer models are supported. See this table for supported models + branches containing fixes for certain models.

```
...

model = TextClassifier(backbone="facebook/bart-large", num_classes=datamodule.num_
↪classes, enable_ort=True)
```

CHAPTER

TWENTYSEVEN

MULTI-LABEL TEXT CLASSIFICATION

27.1 The Task

Multi-label classification is the task of assigning a number of labels from a fixed set to each data point, which can be in any modality (text in this case). Multi-label text classification is supported by the *TextClassifier* via the `multi-label` argument.

27.2 Example

Let's look at the task of classifying comment toxicity. The data we will use in this example is from the kaggle toxic comment classification challenge by jigsaw: www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge. The data is stored in CSV files with this structure:

```
"id","comment_text","toxic","severe_toxic","obscene","threat","insult","identity_hate"
"0000997932d777bf","...",0,0,0,0,0,0
"0002bcb3da6cb337","...",1,1,1,0,1,0
"0005c987bdfc9d4b","...",1,0,0,0,0,0
...
```

Once we've downloaded the data using `download_data()`, we create the *TextClassificationData*. We select a pre-trained backbone to use for our *TextClassifier* and finetune on the toxic comments data. The backbone can be any BERT classification model from HuggingFace/transformers.

Note: When changing the backbone, make sure you pass in the same backbone to the *TextClassifier* and the *TextClassificationData*!

Next, we use the trained *TextClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.text import TextClassificationData, TextClassifier

# 1. Create the DataModule
# Data from the Kaggle Toxic Comment Classification Challenge:
```

(continues on next page)

(continued from previous page)

```
# https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
download_data("https://pl-flash-data.s3.amazonaws.com/jigsaw_toxic_comments.zip", "./data
↪")

datamodule = TextClassificationData.from_csv(
    "comment_text",
    ["toxic", "severe_toxic", "obscene", "threat", "insult", "identity_hate"],
    train_file="data/jigsaw_toxic_comments/train.csv",
    val_split=0.1,
    backbone="unitary/toxic-bert",
)

# 2. Build the task
model = TextClassifier(
    backbone="unitary/toxic-bert",
    num_classes=datamodule.num_classes,
    multi_label=True,
)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=1, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 4. Generate predictions for a few comments!
predictions = model.predict(
    [
        "No, he is an arrogant, self serving, immature idiot. Get it right.",
        "U SUCK HANNAH MONTANA",
        "Would you care to vote? Thx.",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("text_classification_multi_label_model.pt")
```

27.3 Flash Zero

The multi-label text classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash text_classification from_toxic
```

To view configuration options and options for running the text classifier with your own data, use:

```
flash text_classification --help
```

27.4 Serving

The `TextClassifier` is servable. For more information, see *Text Classification*.

CHAPTER

TWENTYEIGHT

QUESTION ANSWERING

28.1 The Task

Question Answering is the task of being able to answer questions pertaining to some known context. For example, given a context about some historical figure, any question pertaininig to the context should be answerable. In our case the article would be our input context and question, and the answer would be the output sequence from the model.

Note: We currently only support Extractive Question Answering, like the task performed using the SQUAD like datasets.

28.2 Example

Let's look at an example. We'll use the SQUAD 2.0 dataset, which contains `train-v2.0.json` and `dev-v2.0.json`. Each JSON file looks like this:

```
{
            "answers": {
                    "answer_start": [94, 87, 94, 94],
                    "text": ["10th and 11th centuries", "in the 10th and 11th centuries",
↪ "10th and 11th centuries", "10th and 11th centuries"]
            },
            "context": "\"The Normans (Norman: Nourmands; French: Normands; Latin:␣
↪Normanni) were the people who in the 10th and 11th centuries gave thei...",
            "id": "56ddde6b9a695914005b9629",
            "question": "When were the Normans in Normandy?",
            "title": "Normans"
    }
...
```

In the above, the `context` key represents the context used for the question and answer, the `question` key represents the question being asked with respect to the context, the `answer` key stores the answer(s) for the question. `id` and `title` are used for unique identification and grouping concepts together respectively. Once we've downloaded the data using `download_data()`, we create the *QuestionAnsweringData*. We select a pre-trained backbone to use for our *QuestionAnsweringTask* and finetune on the SQUAD 2.0 data. The backbone can be any Question Answering model from HuggingFace/transformers.

Note: When changing the backbone, make sure you pass in the same backbone to the *QuestionAnsweringData* and the *QuestionAnsweringTask*!

Next, we use the trained *QuestionAnsweringTask* for inference. Finally, we save the model. Here's the full example:

```
from flash import Trainer
from flash.core.data.utils import download_data
from flash.text import QuestionAnsweringData, QuestionAnsweringTask

# 1. Create the DataModule
download_data("https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v2.0.json", "./
↪data/")
download_data("https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v2.0.json", "./
↪data/")

datamodule = QuestionAnsweringData.from_squad_v2(
    train_file="./data/train-v2.0.json",
    val_file="./data/dev-v2.0.json",
)

# 2. Build the task
model = QuestionAnsweringTask()

# 3. Create the trainer and finetune the model
trainer = Trainer(max_epochs=3, limit_train_batches=1, limit_val_batches=1)
trainer.finetune(model, datamodule=datamodule)

# 4. Answer some Questions!
predictions = model.predict(
    {
        "id": ["56ddde6b9a695914005b9629", "56ddde6b9a695914005b9628"],
        "context": [
            """
        The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the␣
↪people who in the 10th
        and 11th centuries gave their name to Normandy, a region in France. They were␣
↪descended from Norse
        ("Norman" comes from "Norseman") raiders and pirates from Denmark, Iceland and␣
↪Norway who, under
        their leader Rollo, agreed to swear fealty to King Charles III of West Francia.␣
↪Through generations
        of assimilation and mixing with the native Frankish and Roman-Gaulish␣
↪populations, their
        descendants would gradually merge with the Carolingian-based cultures of West␣
↪Francia. The distinct
        cultural and ethnic identity of the Normans emerged initially in the first half␣
↪of the 10th
        century, and it continued to evolve over the succeeding centuries.
        """,
            """
        The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the␣
↪people who in the 10th
```

(continues on next page)

(continued from previous page)

```
        and 11th centuries gave their name to Normandy, a region in France. They were␣
↪descended from Norse
        ("Norman" comes from "Norseman") raiders and pirates from Denmark, Iceland and␣
↪Norway who, under
        their leader Rollo, agreed to swear fealty to King Charles III of West Francia.␣
↪Through generations
        of assimilation and mixing with the native Frankish and Roman-Gaulish␣
↪populations, their
        descendants would gradually merge with the Carolingian-based cultures of West␣
↪Francia. The distinct
        cultural and ethnic identity of the Normans emerged initially in the first half␣
↪of the 10th
        century, and it continued to evolve over the succeeding centuries.
        """,
        ],
        "question": ["When were the Normans in Normandy?", "In what country is Normandy␣
↪located?"],
    }
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("question_answering_on_sqaud_v2.pt")
```

28.3 Accelerate Training & Inference with Torch ORT

Torch ORT converts your model into an optimized ONNX graph, speeding up training & inference when using NVIDIA or AMD GPUs. Enabling Torch ORT requires a single flag passed to the `QuestionAnsweringTask` once installed. See installation instructions here.

Note: Not all Transformer models are supported. See this table for supported models + branches containing fixes for certain models.

```
...

model = QuestionAnsweringTask(backbone="distilbert-base-uncased", max_answer_length=30,␣
↪enable_ort=True)
```

CHAPTER

TWENTYNINE

SUMMARIZATION

29.1 The Task

Summarization is the task of summarizing text from a larger document/article into a short sentence/description. For example, taking a web article and describing the topic in a short sentence. This task is a subset of Sequence to Sequence tasks, which require the model to generate a variable length sequence given an input sequence. In our case the article would be our input sequence, and the short description/sentence would be the output sequence from the model.

29.2 Example

Let's look at an example. We'll use the XSUM dataset, which contains a `train.csv` and `valid.csv`. Each CSV file looks like this:

```
input,target
"The researchers have sequenced the genome of a strain of bacterium that causes the␣
↪virulent infection...","A team of UK scientists hopes to shed light on the mysteries␣
↪of bleeding canker, a disease that is threatening the nation's horse chestnut trees."
"Knight was shot in the leg by an unknown gunman at Miami's Shore Club where West was␣
↪holding a pre-MTV Awards...",Hip hop star Kanye West is being sued by Death Row␣
↪Records founder Suge Knight over a shooting at a beach party in August 2005.
...
```

In the above, the input column represents the long articles/documents, and the target is the short description used as the target. Once we've downloaded the data using `download_data()`, we create the *SummarizationData*. We select a pre-trained backbone to use for our *SummarizationTask* and finetune on the XSUM data. The backbone can be any Seq2Seq summarization model from HuggingFace/transformers.

Note: When changing the backbone, make sure you pass in the same backbone to the *SummarizationData* and the *SummarizationTask*!

Next, we use the trained *SummarizationTask* for inference. Finally, we save the model. Here's the full example:

```
from flash import Trainer
from flash.core.data.utils import download_data
from flash.text import SummarizationData, SummarizationTask

```

(continues on next page)

(continued from previous page)

```
# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/xsum.zip", "./data/")

datamodule = SummarizationData.from_csv(
    "input",
    "target",
    train_file="data/xsum/train.csv",
    val_file="data/xsum/valid.csv",
)

# 2. Build the task
model = SummarizationTask()

# 3. Create the trainer and finetune the model
trainer = Trainer(max_epochs=3)
trainer.finetune(model, datamodule=datamodule)

# 4. Summarize some text!
predictions = model.predict(
    """
    Camilla bought a box of mangoes with a Brixton Âf10 note, introduced last year to␣
↪try to keep the money of local
    people within the community.The couple were surrounded by shoppers as they walked␣
↪along Electric Avenue.
    They came to Brixton to see work which has started to revitalise the borough.
    It was Charles' first visit to the area since 1996, when he was accompanied by the␣
↪former
    South African president Nelson Mandela.Greengrocer Derek Chong, who has run a stall␣
↪on Electric Avenue
    for 20 years, said Camilla had been ""nice and pleasant"" when she purchased the␣
↪fruit.
    ""She asked me what was nice, what would I recommend, and I said we've got some nice␣
↪mangoes.
    She asked me were they ripe and I said yes - they're from the Dominican Republic.""
    Mr Chong is one of 170 local retailers who accept the Brixton Pound.
    Customers exchange traditional pound coins for Brixton Pounds and then spend them at␣
↪the market
    or in participating shops.
    During the visit, Prince Charles spent time talking to youth worker Marcus West, who␣
↪works with children
    nearby on an estate off Coldharbour Lane. Mr West said:
    ""He's on the level, really down-to-earth. They were very cheery. The prince is a␣
↪lovely man.""
    He added: ""I told him I was working with young kids and he said, 'Keep up all the␣
↪good work.'""
    Prince Charles also visited the Railway Hotel, at the invitation of his charity The␣
↪Prince's Regeneration Trust.
    The trust hopes to restore and refurbish the building,
    where once Jimi Hendrix and The Clash played, as a new community and business centre.
↪"
    """
)
```

(continues on next page)

(continued from previous page)

```
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("summarization_model_xsum.pt")
```

29.3 Flash Zero

The summarization task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash summarization
```

To view configuration options and options for running the summarization task with your own data, use:

```
flash summarization --help
```

29.4 Serving

The *SummarizationTask* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.text import SummarizationTask

model = SummarizationTask.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪summarization_model_xsum.pt")
model.serve()
```

You can now perform inference from your client like this:

```
import requests

text = """
Camilla bought a box of mangoes with a Brixton Â£10 note, introduced last year to try to␣
↪keep the money of local
people within the community.The couple were surrounded by shoppers as they walked along␣
↪Electric Avenue.
They came to Brixton to see work which has started to revitalise the borough.
It was Charles' first visit to the area since 1996, when he was accompanied by the former
South African president Nelson Mandela.Greengrocer Derek Chong, who has run a stall on␣
↪Electric Avenue
for 20 years, said Camilla had been ""nice and pleasant"" when she purchased the fruit.
""She asked me what was nice, what would I recommend, and I said we've got some nice␣
↪mangoes.
She asked me were they ripe and I said yes - they're from the Dominican Republic.""
Mr Chong is one of 170 local retailers who accept the Brixton Pound.
Customers exchange traditional pound coins for Brixton Pounds and then spend them at the␣
↪market
```

(continues on next page)

(continued from previous page)

```
or in participating shops.
During the visit, Prince Charles spent time talking to youth worker Marcus West, who␣
↪works with children
nearby on an estate off Coldharbour Lane. Mr West said:
""He's on the level, really down-to-earth. They were very cheery. The prince is a lovely␣
↪man.""
He added: ""I told him I was working with young kids and he said, 'Keep up all the good␣
↪work.'""
Prince Charles also visited the Railway Hotel, at the invitation of his charity The␣
↪Prince's Regeneration Trust.
The trust hopes to restore and refurbish the building,
where once Jimi Hendrix and The Clash played, as a new community and business centre."
"""
body = {"session": "UUID", "payload": {"inputs": {"data": text}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)

print(resp.json())
```

29.5 Accelerate Training & Inference with Torch ORT

Torch ORT converts your model into an optimized ONNX graph, speeding up training & inference when using NVIDIA or AMD GPUs. Enabling Torch ORT requires a single flag passed to the `SummarizationTask` once installed. See installation instructions here.

Note: Not all Transformer models are supported. See this table for supported models + branches containing fixes for certain models.

```
...

model = SummarizationTask(backbone="t5-large", num_classes=datamodule.num_classes,␣
↪enable_ort=True)
```

CHAPTER

THIRTY

TRANSLATION

30.1 The Task

Translation is the task of translating text from a source language to another, such as English to Romanian. This task is a subset of Sequence to Sequence tasks, which requires the model to generate a variable length sequence given an input sequence. In our case, the task will take an English sequence as input, and output the same sequence in Romanian.

30.2 Example

Let's look at an example. We'll use WMT16 English/Romanian, a dataset of English to Romanian samples, based on the Europarl corpora. The data set contains a `train.csv` and `valid.csv`. Each CSV file looks like this:

```
input,target
"Written statements and oral questions (tabling): see Minutes","Declarații scrise și␣
↪întrebări orale (depunere): consultați procesul-verbal"
"Closure of sitting","Ridicarea ședinței"
...
```

In the above the input/target columns represent the English and Romanian translation respectively. Once we've downloaded the data using `download_data()`, we create the *TranslationData*. We select a pre-trained backbone to use for our *TranslationTask* and finetune on the WMT16 data. The backbone can be any Seq2Seq translation model from HuggingFace/transformers.

Note: When changing the backbone, make sure you pass in the same backbone to the *TranslationData* and the *TranslationTask*!

Next, we use the trained *TranslationTask* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.text import TranslationData, TranslationTask

# 1. Create the DataModule
download_data("https://pl-flash-data.s3.amazonaws.com/wmt_en_ro.zip", "./data")
```

(continues on next page)

(continued from previous page)

```
datamodule = TranslationData.from_csv(
    "input",
    "target",
    train_file="data/wmt_en_ro/train.csv",
    val_file="data/wmt_en_ro/valid.csv",
    backbone="Helsinki-NLP/opus-mt-en-ro",
)

# 2. Build the task
model = TranslationTask(backbone="Helsinki-NLP/opus-mt-en-ro")

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.finetune(model, datamodule=datamodule)

# 4. Translate something!
predictions = model.predict(
    [
        "BBC News went to meet one of the project's first graduates.",
        "A recession has come as quickly as 11 months after the first rate hike and as␣
↪long as 86 months.",
        "Of course, it's still early in the election cycle.",
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("translation_model_en_ro.pt")
```

30.3 Flash Zero

The translation task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash translation
```

To view configuration options and options for running the translation task with your own data, use:

```
flash translation --help
```

30.4 Serving

The *TranslationTask* is servable. This means you can call `.serve` to serve your *Task*. Here's an example:

```
from flash.text import TranslationTask

model = TranslationTask.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪translation_model_en_ro.pt")
model.serve()
```

You can now perform inference from your client like this:

```
import requests

text = "Some English text"
body = {"session": "UUID", "payload": {"inputs": {"data": text}}}
resp = requests.post("http://127.0.0.1:8000/predict", json=body)

print(resp.json())
```

30.5 Accelerate Training & Inference with Torch ORT

Torch ORT converts your model into an optimized ONNX graph, speeding up training & inference when using NVIDIA or AMD GPUs. Enabling Torch ORT requires a single flag passed to the `TranslationTask` once installed. See installation instructions here.

Note: Not all Transformer models are supported. See this table for supported models + branches containing fixes for certain models.

```
...

model = TranslationTask(backbone="t5-large", num_classes=datamodule.num_classes, enable_
↪ort=True)
```

CHAPTER

THIRTYONE

POINT CLOUD SEGMENTATION

31.1 The Task

A Point Cloud is a set of data points in space, usually describes by `x`, `y` and `z` coordinates.

PointCloud Segmentation is the task of performing classification at a point-level, meaning each point will associated to a given class. The current integration builds on top Open3D-ML.

31.2 Example

Let's look at an example using a data set generated from the KITTI Vision Benchmark. The data are a tiny subset of the original dataset and contains sequences of point clouds. The data contains multiple folder, one for each sequence and a meta.yaml file describing the classes and their official associated color map. A sequence should contain one folder for scans and one folder for labels, plus a `pose.txt` to re-align the sequence if required. Here's the structure:

```
data
├── meta.yaml
├── 00
│   ├── scans
│   │   ├── 00000.bin
│   │   ├── 00001.bin
│   │   ...
│   ├── labels
│   │   ├── 00000.label
│   │   ├── 00001.label
│   │   ...
│   ├── pose.txt
│   ...
│
└── XX
    ├── scans
    │   ├── 00000.bin
    │   ├── 00001.bin
    │   ...
    ├── labels
    │   ├── 00000.label
    │   ├── 00001.label
```

(continues on next page)

(continued from previous page)

```
|   ...
├── pose.txt
```

Learn more: http://www.semantic-kitti.org/dataset.html

Once we've downloaded the data using `download_data()`, we create the `PointCloudSegmentationData`. We select a pre-trained `randlanet_semantic_kitti` backbone for our `PointCloudSegmentation` task. We then use the trained `PointCloudSegmentation` for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.pointcloud import PointCloudSegmentation, PointCloudSegmentationData

# 1. Create the DataModule
# Dataset Credit: http://www.semantic-kitti.org/
download_data("https://pl-flash-data.s3.amazonaws.com/SemanticKittiTiny.zip", "data/")

datamodule = PointCloudSegmentationData.from_folders(
    train_folder="data/SemanticKittiTiny/train",
    val_folder="data/SemanticKittiTiny/val",
)

# 2. Build the task
model = PointCloudSegmentation(backbone="randlanet_semantic_kitti", num_
↪classes=datamodule.num_classes)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(
    max_epochs=1, limit_train_batches=1, limit_val_batches=1, num_sanity_val_steps=0,␣
↪gpus=torch.cuda.device_count()
)
trainer.fit(model, datamodule)

# 4. Predict what's within a few PointClouds?
predictions = model.predict(
    [
        "data/SemanticKittiTiny/predict/000000.bin",
        "data/SemanticKittiTiny/predict/000001.bin",
    ]
)

# 5. Save the model!
trainer.save_checkpoint("pointcloud_segmentation_model.pt")
```

31.3 Flash Zero

The point cloud segmentation task can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash pointcloud_segmentation
```

To view configuration options and options for running the point cloud segmentation task with your own data, use:

```
flash pointcloud_segmentation --help
```

CHAPTER

THIRTYTWO

POINT CLOUD OBJECT DETECTION

32.1 The Task

A Point Cloud is a set of data points in space, usually describes by `x`, `y` and `z` coordinates.

PointCloud Object Detection is the task of identifying 3D objects in point clouds and their associated classes and 3D bounding boxes.

The current integration builds on top Open3D-ML.

32.2 Example

Let's look at an example using a data set generated from the KITTI Vision Benchmark. The data are a tiny subset of the original dataset and contains sequences of point clouds.

The data contains:

- one folder for scans
- one folder for scan calibrations
- one folder for labels
- a meta.yaml file describing the classes and their official associated color map.

Here's the structure:

```
data
├── meta.yaml
├── train
|   ├── scans
|   |   ├── 00000.bin
|   |   ├── 00001.bin
|   |   ...
|   ├── calibs
|   |   ├── 00000.txt
|   |   ├── 00001.txt
|   |  ...
|   ├── labels
|   |   ├── 00000.txt
|   |   ├── 00001.txt
```

(continues on next page)

(continued from previous page)

```
│   ...
├── val
│   ...
├── predict
    ├── scans
    │   ├── 00000.bin
    │   ├── 00001.bin
    │
    ├── calibs
    │   ├── 00000.txt
    │   ├── 00001.txt
    ├── meta.yaml
```

Learn more: http://www.semantic-kitti.org/dataset.html

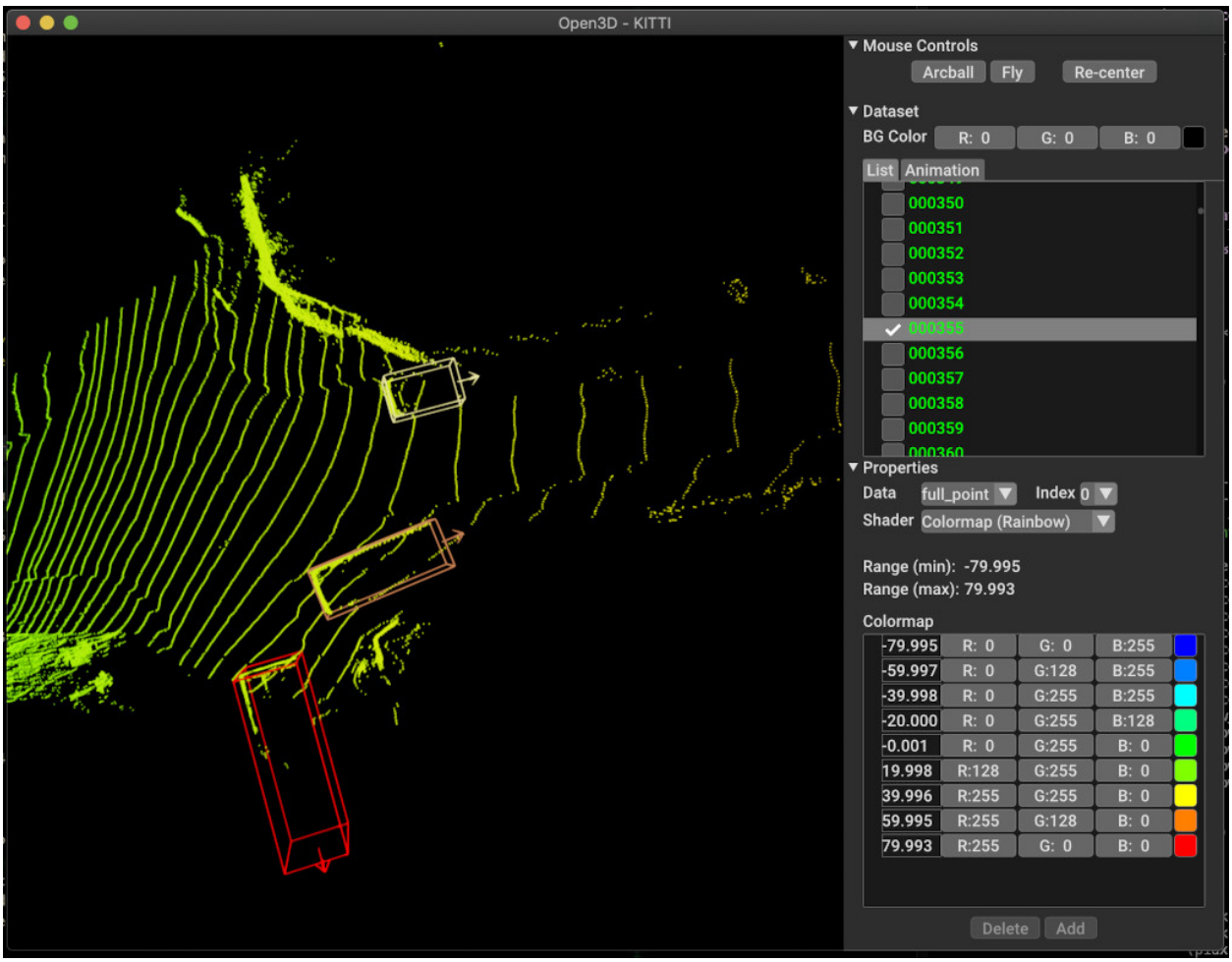
Once we've downloaded the data using `download_data()`, we create the `PointCloudObjectDetectorData`. We select a pre-trained `randlanet_semantic_kitti` backbone for our `PointCloudObjectDetector` task. We then use the trained `PointCloudObjectDetector` for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.data.utils import download_data
from flash.pointcloud import PointCloudObjectDetector, PointCloudObjectDetectorData

# 1. Create the DataModule
# Dataset Credit: http://www.semantic-kitti.org/
download_data("https://pl-flash-data.s3.amazonaws.com/KITTI_tiny.zip", "data/")

datamodule = PointCloudObjectDetectorData.from_folders(
    train_folder="data/KITTI_Tiny/Kitti/train",
    val_folder="data/KITTI_Tiny/Kitti/val",
)

# 2. Build the task
model = PointCloudObjectDetector(backbone="pointpillars_kitti", num_classes=datamodule.
↪num_classes)

# 3. Create the trainer and finetune the model
trainer = flash.Trainer(
    max_epochs=1, limit_train_batches=1, limit_val_batches=1, num_sanity_val_steps=0,␣
↪gpus=torch.cuda.device_count()
)
trainer.fit(model, datamodule)

# 4. Predict what's within a few PointClouds?
predictions = model.predict(
    [
        "data/KITTI_Tiny/Kitti/predict/scans/000000.bin",
        "data/KITTI_Tiny/Kitti/predict/scans/000001.bin",
    ]
)
```

(continues on next page)

(continued from previous page)

```
# 5. Save the model!
trainer.save_checkpoint("pointcloud_detection_model.pt")
```



32.3 Flash Zero

The point cloud object detector can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash pointcloud_detection
```

To view configuration options and options for running the point cloud object detector with your own data, use:

```
flash pointcloud_detection --help
```

CHAPTER

THIRTYTHREE

GRAPH CLASSIFICATION

33.1 The Task

This task consist on classifying graphs. The task predicts which 'class' the graph belongs to. A class is a label that indicates the kind of graph. For example, a label may indicate whether one molecule interacts with another.

The *GraphClassifier* and *GraphClassificationData* classes internally rely on pytorch-geometric.

33.2 Example

Let's look at the task of classifying graphs from the KKI data set from TU Dortmund University.

Once we've created the TUDataset, we create the *GraphClassificationData*. We then create our *GraphClassifier* and train on the KKI data. Next, we use the trained *GraphClassifier* for inference. Finally, we save the model. Here's the full example:

```
import torch

import flash
from flash.core.utilities.imports import example_requires
from flash.graph import GraphClassificationData, GraphClassifier

example_requires("graph")

from torch_geometric.datasets import TUDataset  # noqa: E402

# 1. Create the DataModule
dataset = TUDataset(root="data", name="KKI")

datamodule = GraphClassificationData.from_datasets(
    train_dataset=dataset,
    val_split=0.1,
)

# 2. Build the task
model = GraphClassifier(num_features=datamodule.num_features, num_classes=datamodule.num_
↪classes)
```

(continues on next page)

(continued from previous page)

```
# 3. Create the trainer and fit the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.fit(model, datamodule=datamodule)

# 4. Classify some graphs!
predictions = model.predict(dataset[:3])
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("graph_classification.pt")
```

33.3 Flash Zero

The graph classifier can be used directly from the command line with zero code using *Flash Zero*. You can run the above example with:

```
flash graph_classification
```

To view configuration options and options for running the graph classifier with your own data, use:

```
flash graph_classification --help
```

CHAPTER

THIRTYFOUR

PROVIDERS

Flash is a framework integrator. We rely on many open source frameworks for our tasks, visualizations and backbones. Here's a list of some of the providers we use for backbones and heads within Flash (check them out and star their repos to spread the open source love!):

- airctic/IceVision (https://github.com/airctic/icevision)
- Facebook Research/dino (https://github.com/facebookresearch/dino)
- Facebook Research/PyTorchVideo (https://github.com/facebookresearch/pytorchvideo)
- Hugging Face/transformers (https://github.com/huggingface/transformers)
- Intelligent Systems Lab Org/Open3D-ML (https://github.com/isl-org/Open3D-ML)
- OpenMMLab/MMDetection (https://github.com/open-mmlab/mmdetection)
- pystiche/pystiche (https://github.com/pystiche/pystiche)
- PyTorch/fairseq (https://github.com/pytorch/fairseq)
- PyTorch/torchvision (https://github.com/pytorch/vision)
- qubvel/segmentation_models.pytorch (https://github.com/qubvel/segmentation_models.pytorch)
- rwightman/efficientdet-pytorch (https://github.com/rwightman/efficientdet-pytorch)
- rwightman/pytorch-image-models (https://github.com/rwightman/pytorch-image-models)
- Ultralytics/YOLOV5 (https://github.com/ultralytics/yolov5)

You can also read our guides for some of our larger integrations:

- *FiftyOne*

CHAPTER

THIRTYFIVE

FIFTYONE

We have collaborated with the team at Voxel51 to integrate their tool, FiftyOne, into Lightning Flash.

FiftyOne is an open-source tool for building high-quality datasets and computer vision models. The FiftyOne API and App enable you to visualize datasets and interpret models faster and more effectively.

This integration allows you to view predictions generated by your tasks in the FiftyOne App, as well as easily incorporate FiftyOne Datasets into your tasks. All image and video tasks are supported!

35.1 Installation

In order to utilize this integration, you will need to install FiftyOne:

```
pip install fiftyone
```

35.2 Visualizing Flash predictions

This section shows you how to augment your existing Lightning Flash workflows with a couple of lines of code that let you visualize predictions in FiftyOne. You can visualize predictions for classification, object detection, and semantic segmentation tasks. Doing so is as easy as updating your model to use one of the following serializers:

- *FiftyOneLabels(return_filepath=True)*
- *FiftyOneSegmentationLabels(return_filepath=True)*
- *FiftyOneDetectionLabels(return_filepath=True)*

The `visualize()` function then lets you visualize your predictions in the FiftyOne App. This function accepts a list of dictionaries containing FiftyOne Label objects and filepaths, which is exactly the output of the FiftyOne serializers when the `return_filepath=True` option is specified.

```
from itertools import chain

import torch

import flash
from flash.core.classification import FiftyOneLabels, Labels
from flash.core.data.utils import download_data
from flash.core.finetuning import FreezeUnfreeze
from flash.core.integrations.fiftyone import visualize
from flash.image import ImageClassificationData, ImageClassifier
```

(continues on next page)

(continued from previous page)

```
# 1 Download data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip")

# 2 Load data
datamodule = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
    predict_folder="data/hymenoptera_data/predict/",
)

# 3 Fine tune a model
model = ImageClassifier(
    backbone="resnet18",
    num_classes=datamodule.num_classes,
    serializer=Labels(),
)
trainer = flash.Trainer(
    max_epochs=1,
    gpus=torch.cuda.device_count(),
    limit_train_batches=1,
    limit_val_batches=1,
)
trainer.finetune(
    model,
    datamodule=datamodule,
    strategy=FreezeUnfreeze(unfreeze_epoch=1),
)
trainer.save_checkpoint("image_classification_model.pt")

# 4 Predict from checkpoint
model = ImageClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪image_classification_model.pt")
model.serializer = FiftyOneLabels(return_filepath=True)  # output FiftyOne format
predictions = trainer.predict(model, datamodule=datamodule)
predictions = list(chain.from_iterable(predictions))  # flatten batches

# 5 Visualize predictions in FiftyOne App
# Optional: pass `wait=True` to block execution until App is closed
session = visualize(predictions)
```

The `visualize()` function can be used in all of the following environments:

- **Local Python shell**: The App will launch in a new tab in your default web browser
- **Remote Python shell**: Pass the `remote=True` option and then follow the instructions printed to your remote shell to open the App in your browser on your local machine
- **Jupyter notebook**: The App will launch in the output of your current cell
- **Google Colab**: The App will launch in the output of your current cell
- **Python script**: Pass the `wait=True` option to block execution of your script until the App is closed

See this page for more information about using the FiftyOne App in different environments.

35.3 Using FiftyOne datasets

The above workflow is great for visualizing model predictions. However, if you store your data in a FiftyOne Dataset initially, then you can also visualize ground truth annotations. This allows you to perform more complex analysis with views into your data and evaluation of your model results.

The *from_fiftyone()* method allows you to load your FiftyOne datasets directly into a *DataModule* to be used for training, testing, or inference.

```
from itertools import chain

import fiftyone as fo
import torch

import flash
from flash.core.classification import FiftyOneLabels, Labels
from flash.core.data.utils import download_data
from flash.core.finetuning import FreezeUnfreeze
from flash.image import ImageClassificationData, ImageClassifier

# 1 Download data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip")

# 2 Load data into FiftyOne
train_dataset = fo.Dataset.from_dir(
    dataset_dir="data/hymenoptera_data/train/",
    dataset_type=fo.types.ImageClassificationDirectoryTree,
)
val_dataset = fo.Dataset.from_dir(
    dataset_dir="data/hymenoptera_data/val/",
    dataset_type=fo.types.ImageClassificationDirectoryTree,
)
test_dataset = fo.Dataset.from_dir(
    dataset_dir="data/hymenoptera_data/test/",
    dataset_type=fo.types.ImageClassificationDirectoryTree,
)

# 3 Load data into Flash
datamodule = ImageClassificationData.from_fiftyone(
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    test_dataset=test_dataset,
)

# 4 Fine tune model
model = ImageClassifier(
    backbone="resnet18",
    num_classes=datamodule.num_classes,
    serializer=Labels(),
)
trainer = flash.Trainer(
    max_epochs=1,
    gpus=torch.cuda.device_count(),
```

(continues on next page)

(continued from previous page)

```
    limit_train_batches=1,
    limit_val_batches=1,
)
trainer.finetune(
    model,
    datamodule=datamodule,
    strategy=FreezeUnfreeze(unfreeze_epoch=1),
)
trainer.save_checkpoint("image_classification_model.pt")

# 5 Predict from checkpoint on data with ground truth
model = ImageClassifier.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪image_classification_model.pt")
model.serializer = FiftyOneLabels(return_filepath=False)  # output FiftyOne format
datamodule = ImageClassificationData.from_fiftyone(predict_dataset=test_dataset)
predictions = trainer.predict(model, datamodule=datamodule)
predictions = list(chain.from_iterable(predictions))  # flatten batches

# 6 Add predictions to dataset
test_dataset.set_values("predictions", predictions)

# 7 Evaluate your model
results = test_dataset.evaluate_classifications("predictions", gt_field="ground_truth",␣
↪eval_key="eval")
results.print_report()
plot = results.plot_confusion_matrix()
plot.show()

# 8 Visualize results in the App
session = fo.launch_app(test_dataset)

# Optional: block execution until App is closed
session.wait()
```

35.4 Visualizing embeddings

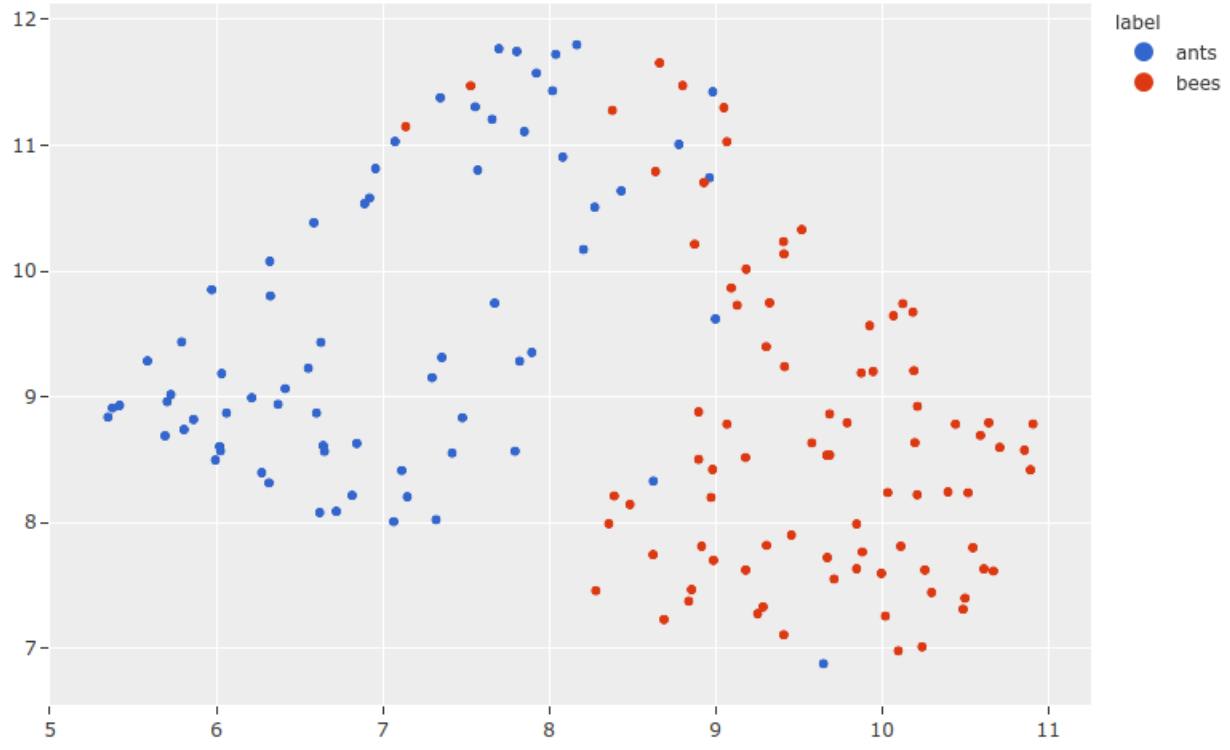
FiftyOne provides the methods for dimensionality reduction and interactive plotting. When combined with *embedding tasks* in Flash, you can accomplish powerful workflows like clustering, similarity search, pre-annotation, and more in only a few lines of code.

```
import fiftyone as fo
import fiftyone.brain as fob
import numpy as np

from flash.core.data.utils import download_data
from flash.image import ImageEmbedder

# 1 Download data
download_data("https://pl-flash-data.s3.amazonaws.com/hymenoptera_data.zip")
```

(continues on next page)

(continued from previous page)

```
# 2 Load data into FiftyOne
dataset = fo.Dataset.from_dir(
    "data/hymenoptera_data/test/",
    fo.types.ImageClassificationDirectoryTree,
)

# 3 Load model
embedder = ImageEmbedder(backbone="resnet101")

# 4 Generate embeddings
filepaths = dataset.values("filepath")
embeddings = np.stack(embedder.predict(filepaths))

# 5 Visualize in FiftyOne App
results = fob.compute_visualization(dataset, embeddings=embeddings)
session = fo.launch_app(dataset)
plot = results.visualize(labels="ground_truth.label")
plot.show()

# Optional: block execution until App is closed
session.wait()
```



CHAPTER

THIRTYSIX

ICEVISION

IceVision from airctic is an awesome computer vision framework which offers a curated collection of hundreds of high-quality pre-trained models for: object detection, keypoint detection, and instance segmentation. In Flash, we've integrated the IceVision framework to provide: data loading, augmentation, backbones, and heads. We use IceVision components in our: *object detection*, *instance segmentation*, and *keypoint detection* tasks. Take a look at their documentation and star IceVision on GitHub to spread the open source love!

36.1 IceData

The IceData library is a community driven dataset hub for IceVision. All of the datasets in IceData can be used out of the box with flash using our `.from_folders` methods and the `parser` argument. Take a look at our *Keypoint Detection* page for an example.

36.2 Albumentations with IceVision and Flash

IceVision provides two utilities for using the albumentations library with their models: - the `Adapter` helper class for adapting an any albumentations transform to work with IceVision records, - the `aug_tfms` utility function that returns a standard augmentation recipe to get the most out of your model.

In Flash, we use the `aug_tfms` as default transforms for the: *object detection*, *instance segmentation*, and *keypoint detection* tasks. You can also provide custom transforms from albumentations using the *`IceVisionTransformAdapter`* (which relies on the IceVision `Adapter` underneath). Here's an example:

```
import albumentations as A

from flash.core.integrations.icevision.transforms import IceVisionTransformAdapter
from flash.image import ObjectDetectionData

train_transform = {
    "pre_tensor_transform": IceVisionTransformAdapter([A.HorizontalFlip(), A.
↪Normalize()]),
}

datamodule = ObjectDetectionData.from_coco(
    ...,
    train_transform=train_transform,
)
```

CHAPTER

THIRTYSEVEN

FLASH

DataSource	The DataSource class encapsulates two hooks: load_data and load_sample.
DataModule	A basic DataModule class for all Flash tasks.
FlashCallback	FlashCallback is an extension of pytorch_lightning.callbacks.Callback.
Preprocess	The *Preprocess* encapsulates all the data processing logic that should run before the data is passed to the model.
Postprocess	The *Postprocess* encapsulates all the data processing logic that should run after the model.
Serializer	A *Serializer* encapsulates a single serialize method which is used to convert the model output into the desired output format when predicting.
Task	A general Task.
Trainer	

37.1 DataSource

class flash.core.data.data_source.**DataSource**

The DataSource class encapsulates two hooks: load_data and load_sample.

The *to_datasets()* method can then be used to automatically construct data sets from the hooks.

generate_dataset(*data*, *running_stage*)

Generate a single dataset with the given input to *load_data()* for the given running_stage.

Parameters

- **data** (Optional[~DATA_TYPE]) – The input to *load_data()* to use to create the dataset.
- **running_stage** (RunningStage) – The running_stage for this dataset.

Return type Union[*AutoDataset*, *IterableAutoDataset*, None]

Returns The constructed *BaseAutoDataset*.

static load_data(*data*, *dataset=None*)

Given the data argument, the load_data hook produces a sequence or iterable of samples or sample metadata. The data argument can be anything, but this method should return a sequence or iterable of mappings from string (e.g. “input”, “target”, “bbox”, etc.) to data (e.g. a target value) or metadata (e.g. a

filename). Where possible, any heavy data loading should be performed in *load_sample()*. If the output is an iterable rather than a sequence (that is, it doesn't have length) then the generated dataset will be an `IterableDataset`.

Parameters

- **data** (*~DATA_TYPE*) – The data required to load the sequence or iterable of samples or sample metadata.
- **dataset** (`Optional`[`Any`]) – Overriding methods can optionally include the dataset argument. Any attributes set on the dataset (e.g. `num_classes`) will also be set on the generated dataset.

Return type `Union`[`Sequence`[`Mapping`[`str`, `Any`]], `Iterable`[`Mapping`[`str`, `Any`]]]

Returns A sequence or iterable of samples or sample metadata to be used as inputs to *load_sample()*.

Example:

```
# data: "."
# output: [{"input": "./cat/1.png", "target": 1}, ..., {"input": "./dog/10.png",
↪ "target": 0}]

output: Sequence[Mapping[str, Any]] = load_data(data)
```

static load_sample(*sample*, *dataset=None*)

Given an element from the output of a call to *load_data()*, this hook should load a single data sample. The keys and values in the `sample` argument will be same as the keys and values in the outputs of *load_data()*.

Parameters

- **sample** (`Mapping`[`str`, `Any`]) – An element (sample or sample metadata) from the output of a call to *load_data()*.
- **dataset** (`Optional`[`Any`]) – Overriding methods can optionally include the dataset argument. Any attributes set on the dataset (e.g. `num_classes`) will also be set on the generated dataset.

Return type `Any`

Returns The loaded sample as a mapping with string keys (e.g. "input", "target") that can be processed by the *pre_tensor_transform()*.

Example:

```
# sample: {"input": "./cat/1.png", "target": 1}
# output: {"input": PIL.Image, "target": 1}

output: Mapping[str, Any] = load_sample(sample)
```

to_datasets(*train_data=None*, *val_data=None*, *test_data=None*, *predict_data=None*)

Construct data sets (of type *BaseAutoDataset*) from this data source by calling *load_data()* with each of the `*_data` arguments. If an argument is given as `None` then no dataset will be created for that stage (`train`, `val`, `test`, `predict`).

Parameters

- **train_data** (`Optional`[~DATA_TYPE]) – The input to *load_data()* to use to create the train dataset.

- **val_data** (Optional[~DATA_TYPE]) – The input to *load_data()* to use to create the validation dataset.
- **test_data** (Optional[~DATA_TYPE]) – The input to *load_data()* to use to create the test dataset.
- **predict_data** (Optional[~DATA_TYPE]) – The input to *load_data()* to use to create the predict dataset.

Return type Tuple[Optional[*BaseAutoDataset*], ...]

Returns A tuple of `train_dataset`, `val_dataset`, `test_dataset`, `predict_dataset`. If any `*_data` argument is not passed to this method then the corresponding `*_dataset` will be `None`.

37.2 DataModule

class flash.core.data.data_module.**DataModule**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

A basic DataModule class for all Flash tasks. This class includes references to a *DataSource*, *Preprocess*, *Postprocess*, and a *BaseDataFetcher*.

Parameters

- **train_dataset** (Optional[Dataset]) – Dataset for training. Defaults to None.
- **val_dataset** (Optional[Dataset]) – Dataset for validating model performance during training. Defaults to None.
- **test_dataset** (Optional[Dataset]) – Dataset to test model performance. Defaults to None.
- **predict_dataset** (Optional[Dataset]) – Dataset for predicting. Defaults to None.
- **data_source** (Optional[*DataSource*]) – The *DataSource* that was used to create the datasets.
- **preprocess** (Optional[*Preprocess*]) – The *Preprocess* to use when constructing the *DataPipeline*. If None, a *DefaultPreprocess* will be used.
- **postprocess** (Optional[*Postprocess*]) – The *Postprocess* to use when constructing the *DataPipeline*. If None, a plain *Postprocess* will be used.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to attach to the *Preprocess*. If None, the output from *configure_data_fetcher()* will be used.
- **val_split** (Optional[float]) – An optional float which gives the relative amount of the training dataset to use for the validation dataset.
- **batch_size** (int) – The batch size to be used by the DataLoader. Defaults to 1.
- **num_workers** (Optional[int]) – The number of workers to use for parallelized loading. Defaults to None which equals the number of available CPU threads, or 0 for Windows or Darwin platform.
- **sampler** (Optional[Type[Sampler]]) – A sampler following the Sampler type. Will be passed to the DataLoader for the training dataset. Defaults to None.

available_data_sources()
Get the list of available data source names for use with this *DataModule*.

Return type Sequence[str]

Returns The list of data source names.

static configure_data_fetcher(**args*, ***kwargs*)
This function is used to configure a *BaseDataFetcher*.

Override with your custom one.

Return type *BaseDataFetcher*

classmethod from_csv(*input_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)
Creates a *DataModule* object from the given CSV files using the *DataSource* of name CSV from the passed or constructed *Preprocess*.

Parameters

- **input_fields** (Union[str, Sequence[str]]) – The field or fields (columns) in the CSV file to use for the input.
- **target_fields** (Union[str, Sequence[str], None]) – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (Optional[str]) – The CSV file containing the training data.
- **val_file** (Optional[str]) – The CSV file containing the validation data.
- **test_file** (Optional[str]) – The CSV file containing the testing data.
- **predict_file** (Optional[str]) – The CSV file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.

- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_csv(
    "input",
    "target",
    train_file="train_data.csv",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod from_data_source(*data_source*, *train_data=None*, *val_data=None*, *test_data=None*, *predict_data=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given inputs to *load_data()* (train_data, val_data, test_data, predict_data). The data source will be resolved from the instantiated *Preprocess* using *data_source_of_name()*.

Parameters

- **data_source** (str) – The name of the data source to use for the *load_data()*.
- **train_data** (Optional[Any]) – The input to *load_data()* to use when creating the train dataset.
- **val_data** (Optional[Any]) – The input to *load_data()* to use when creating the validation dataset.
- **test_data** (Optional[Any]) – The input to *load_data()* to use when creating the test dataset.
- **predict_data** (Optional[Any]) – The input to *load_data()* to use when creating the predict dataset.
- **train_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.

- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_data_source(
    DefaultDataSources.FOLDERS,
    train_data="train_folder",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod **from_datasets**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given datasets using the *DataSource* of name DATASETS from the passed or constructed *Preprocess*.

Parameters

- **train_dataset** (Optional[Dataset]) – Dataset used during training.
- **val_dataset** (Optional[Dataset]) – Dataset used during validating.
- **test_dataset** (Optional[Dataset]) – Dataset used during testing.
- **predict_dataset** (Optional[Dataset]) – Dataset used during predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.

- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_datasets(
    train_dataset=train_dataset,
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod from_fiftyone(*cls*, *train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given FiftyOne Datasets using the *DataSource* of name FIFTYONE from the passed or constructed *Preprocess*.

Parameters

- **train_dataset** (None) – The fiftyone.core.collections.SampleCollection containing the train data.
- **val_dataset** (None) – The fiftyone.core.collections.SampleCollection containing the validation data.
- **test_dataset** (None) – The fiftyone.core.collections.SampleCollection containing the test data.
- **predict_dataset** (None) – The fiftyone.core.collections.SampleCollection containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.

- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
train_dataset = fo.Dataset.from_dir(
    "/path/to/dataset",
    dataset_type=fo.types.ImageClassificationDirectoryTree,
)
data_module = DataModule.from_fiftyone(
    train_data = train_dataset,
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod from_files(*train_files=None*, *train_targets=None*, *val_files=None*, *val_targets=None*, *test_files=None*, *test_targets=None*, *predict_files=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given sequences of files using the *DataSource* of name FILES from the passed or constructed *Preprocess*.

Parameters

- **train_files** (Optional[Sequence[str]]) – A sequence of files to use as the train inputs.
- **train_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per train file) to use as the train targets.
- **val_files** (Optional[Sequence[str]]) – A sequence of files to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation file) to use as the validation targets.
- **test_files** (Optional[Sequence[str]]) – A sequence of files to use as the test inputs.

- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test file) to use as the test targets.
- **predict_files** (Optional[Sequence[str]]) – A sequence of files to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The `sampler` to use for the `train_dataloader`.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

classmethod from_folders(*train_folder=None*, *val_folder=None*, *test_folder=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given folders using the *DataSource* of name `FOLDERS` from the passed or constructed *Preprocess*.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.

- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The `sampler` to use for the `train_dataloader`.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

classmethod from_json(*input_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, *field=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given JSON files using the *DataSource* of name `JSON` from the passed or constructed *Preprocess*.

Parameters

- **input_fields** (Union[str, Sequence[str]]) – The field or fields in the JSON objects to use for the input.
- **target_fields** (Union[str, Sequence[str], None]) – The field or fields in the JSON objects to use for the target.
- **train_file** (Optional[str]) – The JSON file containing the training data.
- **val_file** (Optional[str]) – The JSON file containing the validation data.
- **test_file** (Optional[str]) – The JSON file containing the testing data.
- **predict_file** (Optional[str]) – The JSON file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.

- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The `sampler` to use for the `train_dataloader`.
- **field** (Optional[str]) – To specify the field that holds the data in the JSON file.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_json(
    "input",
    "target",
    train_file="train_data.json",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)

# In the case where the data is of the form:
# {
#     "version": 0.0.x,
#     "data": [
#         {
#             "input_field" : "input_data",
#             "target_field" : "target_output"
#         },
#         ...
#     ]
# }

data_module = DataModule.from_json(
    "input",
    "target",
    train_file="train_data.json",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
```

(continues on next page)

(continued from previous page)

```
    feild="data"
)
```

classmethod from_numpy(*train_data=None*, *train_targets=None*, *val_data=None*, *val_targets=None*, *test_data=None*, *test_targets=None*, *predict_data=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given numpy array using the *DataSource* of name `NUMPY` from the passed or constructed *Preprocess*.

Parameters

- **train_data** (Optional[Collection[ndarray]]) – A numpy array to use as the train inputs.
- **train_targets** (Optional[Collection[Any]]) – A sequence of targets (one per train input) to use as the train targets.
- **val_data** (Optional[Collection[ndarray]]) – A numpy array to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation input) to use as the validation targets.
- **test_data** (Optional[Collection[ndarray]]) – A numpy array to use as the test inputs.
- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test input) to use as the test targets.
- **predict_data** (Optional[Collection[ndarray]]) – A numpy array to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The `Preprocess` to pass to the *DataModule*. If `None`, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The `sampler` to use for the `train_dataloader`.

- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_numpy(
    train_files=np.random.rand(3, 128),
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod from_tensors(*train_data=None*, *train_targets=None*, *val_data=None*, *val_targets=None*, *test_data=None*, *test_targets=None*, *predict_data=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given tensors using the *DataSource* of name `TENSOR` from the passed or constructed *Preprocess*.

Parameters

- **train_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the train inputs.
- **train_targets** (Optional[Collection[Any]]) – A sequence of targets (one per train input) to use as the train targets.
- **val_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation input) to use as the validation targets.
- **test_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the test inputs.
- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test input) to use as the test targets.
- **predict_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.

- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_tensors(
    train_files=torch.rand(3, 128),
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

postprocess_cls
alias of *flash.core.data.process.Postprocess*

property predict_dataset: Optional[torch.utils.data.Dataset]
This property returns the predict dataset.

Return type Optional[Dataset]

show_predict_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the predict dataloader.

Return type None

show_test_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the test dataloader.

Return type None

show_train_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the train dataloader.

Return type None

show_val_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the validation dataloader.

Return type None

property test_dataset: Optional[torch.utils.data.Dataset]
This property returns the test dataset.

Return type Optional[Dataset]

property train_dataset: Optional[torch.utils.data.Dataset]
This property returns the train dataset.

Return type Optional[Dataset]

property val_dataset: Optional[torch.utils.data.Dataset]
This property returns the validation dataset.

Return type Optional[Dataset]

37.3 FlashCallback

class flash.core.data.callback.**FlashCallback**(**args*, ***kwargs*)
FlashCallback is an extension of pytorch_lightning.callbacks.Callback.

A callback is a self-contained program that can be reused across projects. Flash and Lightning have a callback system to execute callbacks when needed. Callbacks should capture any NON-ESSENTIAL logic that is NOT required for your lightning module to run.

Same as PyTorch Lightning, Callbacks can be provided directly to the Trainer:

```
trainer = Trainer(callbacks=[MyCustomCallback()])
```

on_collate(*batch*, *running_stage*)
Called once collate has been applied to a sequence of samples.

Return type None

on_load_sample(*sample*, *running_stage*)
Called once a sample has been loaded using load_sample.

Return type None

on_per_batch_transform(*batch*, *running_stage*)
Called once per_batch_transform has been applied to a batch.

Return type None

on_per_batch_transform_on_device(*batch*, *running_stage*)
Called once per_batch_transform_on_device has been applied to a sample.

Return type None

on_per_sample_transform_on_device(*sample*, *running_stage*)
Called once per_sample_transform_on_device has been applied to a sample.

Return type None

on_post_tensor_transform(*sample*, *running_stage*)
Called once post_tensor_transform has been applied to a sample.

Return type None

on_pre_tensor_transform(*sample*, *running_stage*)
Called once pre_tensor_transform has been applied to a sample.

Return type None

on_to_tensor_transform(*sample*, *running_stage*)
Called once to_tensor_transform has been applied to a sample.

Return type None

37.4 Preprocess

class flash.core.data.process.**Preprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_sources=None*, *deserializer=None*, *default_data_source=None*)

The *Preprocess* encapsulates all the data processing logic that should run before the data is passed to the model. It is particularly useful when you want to provide an end to end implementation which works with 4 different stages: `train`, `validation`, `test`, and inference (`predict`).

The *Preprocess* supports the following hooks:

- **`pre_tensor_transform`: Performs transforms on a single data sample.** Example:

```
* Input: Receive a PIL Image and its label.

* Action: Rotate the PIL Image.

* Output: Return the rotated PIL image and its label.
```

- **`to_tensor_transform`: Converts a single data sample to a tensor / data structure containing tensors.** Example:

```
* Input: Receive the rotated PIL Image and its label.

* Action: Convert the rotated PIL Image to a tensor.

* Output: Return the tensored image and its label.
```

- **`post_tensor_transform`: Performs transform on a single tensor sample.** Example:

```
* Input: Receive the tensored image and its label.

* Action: Flip the tensored image randomly.

* Output: Return the tensored image and its label.
```

- **`per_batch_transform`: Performs transforms on a batch.** In this example, we decided not to override the hook.

- **`per_sample_transform_on_device`: Performs transform on a sample already on a `GPU` or `TPU`.** Example:

```
* Input: Receive a tensored image on device and its label.

* Action: Apply random transforms.

* Output: Return an augmented tensored image on device and its label.
```

- **`collate`: Converts a sequence of data samples into a batch.** Defaults to `torch.utils.data._utils.collate.default_collate`. Example:

```
* Input: Receive a list of augmented tensored images and their respective␣
↪labels.
```

(continues on next page)

(continued from previous page)

```
* Action: Collate the list of images into batch.

* Output: Return a batch of images and their labels.
```

- **per_batch_transform_on_device: Performs transform on a batch already on GPU or TPU.** Example:

```
* Input: Receive a batch of images and their labels.

* Action: Apply normalization on the batch by subtracting the mean
    and dividing by the standard deviation from ImageNet.

* Output: Return a normalized augmented batch of images and their labels.
```

Note: The `per_sample_transform_on_device` and `per_batch_transform` are mutually exclusive as it will impact performances.

Data processing can be configured by overriding hooks or through transforms. The preprocess transforms are given as a mapping from hook names to callables. Default transforms can be configured by overriding the `default_transforms` or `{train,val,test,predict}_default_transforms` methods. These can then be overridden by the user with the `{train,val,test,predict}_transform` arguments to the `Preprocess`. All of the hooks can be used in the transform mappings.

Example:

```
class CustomPreprocess(Preprocess):

    def default_transforms() -> Mapping[str, Callable]:
        return {
            "to_tensor_transform": transforms.ToTensor(),
            "collate": torch.utils.data._utils.collate.default_collate,
        }

    def train_default_transforms() -> Mapping[str, Callable]:
        return {
            "pre_tensor_transform": transforms.RandomHorizontalFlip(),
            "to_tensor_transform": transforms.ToTensor(),
            "collate": torch.utils.data._utils.collate.default_collate,
        }
```

When overriding hooks for particular stages, you can prefix with `train`, `val`, `test` or `predict`. For example, you can achieve the same as the above example by implementing `train_pre_tensor_transform` and `train_to_tensor_transform`.

Example:

```
class CustomPreprocess(Preprocess):

    def train_pre_tensor_transform(self, sample: PIL.Image) -> PIL.Image:
        return transforms.RandomHorizontalFlip()(sample)

    def to_tensor_transform(self, sample: PIL.Image) -> torch.Tensor:
```

(continues on next page)

(continued from previous page)

```
        return transforms.ToTensor()(sample)

    def collate(self, samples: List[torch.Tensor]) -> torch.Tensor:
        return torch.utils.data._utils.collate.default_collate(samples)
```

Each hook is aware of the Trainer running stage through booleans. These are useful for adapting functionality for a stage without duplicating code.

Example:

```
class CustomPreprocess(Preprocess):

    def pre_tensor_transform(self, sample: PIL.Image) -> PIL.Image:

        if self.training:
            # logic for training

        elif self.validating:
            # logic for validation

        elif self.testing:
            # logic for testing

        elif self.predicting:
            # logic for predicting
```

available_data_sources()
Get the list of available data source names for use with this *Preprocess*.

Return type Sequence[str]

Returns The list of data source names.

collate(*samples*, *metadata=None*)
Transform to convert a sequence of samples to a collated batch.

Return type Any

data_source_of_name(*data_source_name*)
Get the *DataSource* of the given name from the *Preprocess*.

Parameters **data_source_name** (str) – The name of the data source to look up.

Return type *DataSource*

Returns The *DataSource* of the given name.

Raises **MisconfigurationException** – If the requested data source is not configured by this *Preprocess*.

static default_transforms()
The default transforms to use.

Will be overridden by transforms passed to the __init__.

Return type Optional[Dict[str, Callable]]

per_batch_transform(*batch*)
Transforms to apply to a whole batch (if possible use this for efficiency).

Note: This option is mutually exclusive with *per_sample_transform_on_device()*, since if both are specified, uncollation has to be applied.

Return type Any

per_batch_transform_on_device(*batch*)
Transforms to apply to a whole batch (if possible use this for efficiency).

Note: This function won't be called within the dataloader workers, since to make that happen each of the workers would have to create it's own CUDA-context which would pollute GPU memory (if on GPU).

Return type Any

per_sample_transform_on_device(*sample*)
Transforms to apply to the data before the collation (per-sample basis).

Note: This option is mutually exclusive with *per_batch_transform()*, since if both are specified, uncollation has to be applied.

Note: This function won't be called within the dataloader workers, since to make that happen each of the workers would have to create it's own CUDA-context which would pollute GPU memory (if on GPU).

Return type Any

post_tensor_transform(*sample*)
Transforms to apply on a tensor.

Return type Tensor

pre_tensor_transform(*sample*)
Transforms to apply on a single object.

Return type Any

to_tensor_transform(*sample*)
Transforms to convert single object to a tensor.

Return type Tensor

property transforms: Dict[str, Optional[Dict[str, Callable]]]
The transforms currently being used by this *Preprocess*.

Return type Dict[str, Optional[Dict[str, Callable]]]

37.5 Postprocess

class flash.core.data.process.**Postprocess**(*save_path=None*)

The *Postprocess* encapsulates all the data processing logic that should run after the model.

static per_batch_transform(*batch*)

Transforms to apply on a whole batch before uncollation to individual samples.

Can involve both CPU and Device transforms as this is not applied in separate workers.

Return type Any

static per_sample_transform(*sample*)

Transforms to apply to a single sample after splitting up the batch.

Can involve both CPU and Device transforms as this is not applied in separate workers.

Return type Any

static save_data(*data*, *path*)

Saves all data together to a single path.

Return type None

static save_sample(*sample*, *path*)

Saves each sample individually to a given path.

Return type None

static uncollate(*batch*)

Uncollates a batch into single samples.

Tries to preserve the type whereever possible.

Return type Any

37.6 Serializer

class flash.core.data.process.**Serializer**

A *Serializer* encapsulates a single `serialize` method which is used to convert the model output into the desired output format when predicting.

disable()

Disable serialization.

enable()

Enable serialization.

static serialize(*sample*)

Serialize the given sample into the desired output format.

Parameters **sample** (Any) – The output from the *Postprocess*.

Return type Any

Returns The serialized output.

37.7 Task

class flash.core.model.**Task**(*model=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=5e-05*, *deserializer=None*, *preprocess=None*, *postprocess=None*, *serializer=None*)

A general Task.

Parameters

- **model** (Optional[Module]) – Model to use for the task.
- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training, defaults to torch.optim.Adam.
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **learning_rate** (float) – Learning rate to use for training, defaults to 5e-5.
- **preprocess** (Optional[*Preprocess*]) – *Preprocess* to use as the default for this task.
- **postprocess** (Optional[*Postprocess*]) – *Postprocess* to use as the default for this task.

static apply_filtering(*y*, *y_hat*)

This function is used to filter some labels or predictions which aren't conform.

Return type Tuple[Tensor, Tensor]

build_data_pipeline(*data_source=None*, *deserializer=None*, *data_pipeline=None*)

Build a *DataPipeline* incorporating available *Preprocess* and *Postprocess* objects. These will be overridden in the following resolution order (lowest priority first):

- Lightning Datamodule, either attached to the *Trainer* or to the *Task*.
- *Task* defaults given to Task.__init__().
- *Task* manual overrides by setting data_pipeline.
- *DataPipeline* passed to this method.

Parameters **data_pipeline** (Optional[*DataPipeline*]) – Optional highest priority source of *Preprocess* and *Postprocess*.

Return type Optional[*DataPipeline*]

Returns The fully resolved *DataPipeline*.

get_num_training_steps()

Total training steps inferred from datamodule and devices.

Return type int

predict(*x*, *data_source=None*, *deserializer=None*, *data_pipeline=None*)

Predict function for raw data or processed data.

Parameters

- **x** (Any) – Input to predict. Can be raw data or processed data. If str, assumed to be a folder of data.

- **data_pipeline** (Optional[*DataPipeline*]) – Use this to override the current data pipeline

Return type Any

Returns The post-processed model predictions

step(*batch*, *batch_idx*, *metrics*)
The training/validation/test step.

Override for custom behavior.

Return type Any

37.8 Trainer

class flash.core.trainer.**Trainer**(**args*, *serve_sanity_check=False*, ***kwargs*)

classmethod add_argparse_args(**args*, ***kwargs*)
See pytorch_lightning.utilities.argparse.add_argparse_args().

Return type ArgumentParser

finetune(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*, *strategy=None*)
Runs the full optimization routine. Same as pytorch_lightning.Trainer.fit(), but unfreezes layers of the backbone throughout training layers of the backbone throughout training.

Parameters

- **datamodule** (Optional[LightningDataModule]) – A instance of LightningDataModule.
- **model** (LightningModule) – Model to fit.
- **train_dataloader** (Optional[DataLoader]) – A PyTorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **val_dataloaders** (Union[DataLoader, List[DataLoader], None]) – Either a single PyTorch Dataloader or a list of them, specifying validation samples. If the model has a predefined val_dataloaders method this will be skipped
- **strategy** (Union[str, BaseFinetuning, None]) – Should either be a string or a fine-tuning callback subclassing pytorch_lightning.callbacks.BaseFinetuning.

  Default strategies can be enabled with these strings:

  - "no_freeze",
  - "freeze",
  - "freeze_unfreeze",
  - "unfreeze_milestones".

fit(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*)
Runs the full optimization routine. Same as pytorch_lightning.Trainer.fit()

Parameters

- **datamodule** (Optional[LightningDataModule]) – A instance of LightningDataModule.
- **model** (LightningModule) – Model to fit.

- **train_dataloader** (Optional[DataLoader]) – A Pytorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **val_dataloaders** (Union[DataLoader, List[DataLoader], None]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples. If the model has a predefined val_dataloaders method this will be skipped

classmethod from_argparse_args(*args*, ***kwargs*)

Modified version of pytorch_lightning.utilities.argparse.from_argparse_args() which populates valid_kwargs from pytorch_lightning.Trainer.

Return type *Trainer*

request_dataloader(**args*, ***kwargs*)

Handles downloading data in the GPU or TPU case.

Return type Union[DataLoader, List[DataLoader]]

Returns The dataloader

CHAPTER

THIRTYEIGHT

FLASH.CORE

38.1 flash.core.adapter

Adapter	The `Adapter` is a lightweight interface that can be used to encapsulate the logic from a particular provider within a *Task*.
AdapterTask	The `AdapterTask` is a *Task* which wraps an *Adapter* and forwards all of the hooks.

38.1.1 Adapter

class flash.core.adapter.**Adapter**(**args: Any*, ***kwargs: Any*)

The `Adapter` is a lightweight interface that can be used to encapsulate the logic from a particular provider within a *Task*.

abstract classmethod from_task(*task*, ***kwargs*)

Instantiate the adapter from the given *Task*.

This includes resolution / creation of backbones / heads and any other provider specific options.

38.1.2 AdapterTask

class flash.core.adapter.**AdapterTask**(*adapter*, ***kwargs*)

The AdapterTask is a *Task* which wraps an *Adapter* and forwards all of the hooks.

Parameters

- **adapter** (*Adapter*) – The *Adapter* to wrap.
- **kwargs** – Keyword arguments to be passed to the base *Task*.

38.2 flash.core.classification

Classes	A *Serializer* which applies an argmax to the model outputs (either logits or probabilities) and converts to a list.
ClassificationSerializer	A base class for classification serializers.
ClassificationTask	
FiftyOneLabels	A *Serializer* which converts the model outputs to FiftyOne classification format.
Labels	A *Serializer* which converts the model outputs (either logits or probabilities) to the label of the argmax classification.
Logits	A *Serializer* which simply converts the model outputs (assumed to be logits) to a list.
PredsClassificationSerializer	A *ClassificationSerializer* which gets the PREDS from the sample.
Probabilities	A *Serializer* which applies a softmax to the model outputs (assumed to be logits) and converts to a list.

38.2.1 Classes

class flash.core.classification.**Classes**(*multi_label=False*, *threshold=0.5*)

A *Serializer* which applies an argmax to the model outputs (either logits or probabilities) and converts to a list.

Parameters

- **multi_label** (bool) – If true, treats outputs as multi label logits.
- **threshold** (float) – The threshold to use for multi_label classification.

38.2.2 ClassificationSerializer

class flash.core.classification.**ClassificationSerializer**(*multi_label=False*)
A base class for classification serializers.

Parameters **multi_label**¶ (bool) – If true, treats outputs as multi label logits.

38.2.3 ClassificationTask

class flash.core.classification.**ClassificationTask**(**args*, *num_classes=None*, *loss_fn=None*, *metrics=None*, *multi_label=False*, *serializer=None*, ***kwargs*)

38.2.4 FiftyOneLabels

class flash.core.classification.**FiftyOneLabels**(*labels=None*, *multi_label=False*, *threshold=None*, *store_logits=False*, *return_filepath=False*)
A *Serializer* which converts the model outputs to FiftyOne classification format.

Parameters

- **labels**¶ (Optional[List[str]]) – A list of labels, assumed to map the class index to the label for that class. If labels is not provided, will attempt to get them from the *LabelsState*.
- **multi_label**¶ (bool) – If true, treats outputs as multi label logits.
- **threshold**¶ (Optional[float]) – A threshold to use to filter candidate labels. In the single label case, predictions below this threshold will be replaced with None
- **store_logits**¶ (bool) – Boolean determining whether to store logits in the FiftyOne labels
- **return_filepath**¶ (bool) – Boolean determining whether to return a dict containing filepath and FiftyOne labels (True) or only a list of FiftyOne labels (False)

38.2.5 Labels

class flash.core.classification.**Labels**(*labels=None*, *multi_label=False*, *threshold=0.5*)
A *Serializer* which converts the model outputs (either logits or probabilities) to the label of the argmax classification.

Parameters

- **labels**¶ (Optional[List[str]]) – A list of labels, assumed to map the class index to the label for that class. If labels is not provided, will attempt to get them from the *LabelsState*.
- **multi_label**¶ (bool) – If true, treats outputs as multi label logits.
- **threshold**¶ (float) – The threshold to use for multi_label classification.

38.2.6 Logits

class flash.core.classification.**Logits**(*multi_label=False*)
A *Serializer* which simply converts the model outputs (assumed to be logits) to a list.

38.2.7 PredsClassificationSerializer

class flash.core.classification.**PredsClassificationSerializer**(*multi_label=False*)
A *ClassificationSerializer* which gets the PREDS from the sample.

38.2.8 Probabilities

class flash.core.classification.**Probabilities**(*multi_label=False*)
A *Serializer* which applies a softmax to the model outputs (assumed to be logits) and converts to a list.

38.3 flash.core.finetuning

FlashBaseFinetuning	FlashBaseFinetuning can be used to create a custom Flash Finetuning Callback.
FreezeUnfreeze	
NoFreeze	
UnfreezeMilestones	

38.3.1 FlashBaseFinetuning

class flash.core.finetuning.**FlashBaseFinetuning**(*attr_names='backbone'*, *train_bn=True*)
FlashBaseFinetuning can be used to create a custom Flash Finetuning Callback.

Override `finetune_function()` to put your unfreeze logic.

38.3.2 FreezeUnfreeze

class flash.core.finetuning.**FreezeUnfreeze**(*attr_names='backbone'*, *train_bn=True*, *unfreeze_epoch=10*)

38.3.3 NoFreeze

class flash.core.finetuning.**NoFreeze**(**args*, ***kwargs*)

38.3.4 UnfreezeMilestones

class flash.core.finetuning.**UnfreezeMilestones**(*attr_names='backbone'*, *train_bn=True*, *unfreeze_milestones=(5, 10)*, *num_layers=5*)

38.4 flash.core.integrations.fiftyone

visualize	Visualizes predictions from a model with a FiftyOne Serializer in the FiftyOne App.

38.4.1 flash.core.integrations.fiftyone.utils.visualize

flash.core.integrations.fiftyone.utils.**visualize**(*predictions*, *filepaths=None*, *label_field='predictions'*, *wait=False*, ***kwargs*)

Visualizes predictions from a model with a FiftyOne Serializer in the FiftyOne App.

This method can be used in all of the following environments:

- **Local Python shell**: The App will launch in a new tab in your default web browser.
- **Remote Python shell**: Pass the `remote=True` option to this method and then follow the instructions printed to your remote shell to open the App in your browser on your local machine.
- **Jupyter notebook**: The App will launch in the output of your current cell.
- **Google Colab**: The App will launch in the output of your current cell.
- **Python script**: Pass the `wait=True` option to block execution of your script until the App is closed.

See this page for more information about using the FiftyOne App in different environments.

Parameters

- **predictions** (Union[List[None], List[Dict[str, None]]]) – Can be either a list of FiftyOne labels that will be matched with the corresponding `filepaths`, or a list of dictionaries with "filepath" and "predictions" keys that contains the filepaths and predictions.
- **filepaths** (Optional[List[str]]) – A list of filepaths to images or videos corresponding to the provided `predictions`.
- **label_field** (Optional[str]) – The name of the label field in which to store the predictions in the FiftyOne dataset.
- **wait** (Optional[bool]) – Whether to block execution until the FiftyOne App is closed.
- ****kwargs** – Optional keyword arguments for `fiftyone.core.session.launch_app`.

Return type None

Returns a fiftyone.core.session.Session

38.5 flash.core.integrations.icevision

IceVisionTransformAdapter

default_transforms	The default transforms from IceVision.
train_default_transforms	The default augmentations from IceVision.

38.5.1 flash.core.integrations.icevision.transforms.IceVisionTransformAdapter

class flash.core.integrations.icevision.transforms.**IceVisionTransformAdapter**(**args: Any*, ***kwargs: Any*)

__init__(*transform*)

Methods

__init__(transform)
forward(x)

38.5.2 flash.core.integrations.icevision.transforms.default_transforms

flash.core.integrations.icevision.transforms.**default_transforms**(*image_size*)
The default transforms from IceVision.

Return type Dict[str, Callable]

38.5.3 flash.core.integrations.icevision.transforms.train_default_transforms

flash.core.integrations.icevision.transforms.**train_default_transforms**(*image_size*)

The default augmentations from IceVision.

Return type Dict[str, Callable]

38.6 flash.core.model

BenchmarkConvergenceCI	
CheckDependenciesMeta	
ModuleWrapperBase	The ModuleWrapperBase is a base for classes which wrap a LightningModule or an instance of ModuleWrapperBase.
DatasetProcessor	The DatasetProcessor mixin provides hooks for classes which need custom logic for producing the data loaders for each running stage given the corresponding dataset.
Task	A general Task.

38.6.1 BenchmarkConvergenceCI

class flash.core.model.**BenchmarkConvergenceCI**(**args: Any*, ***kwargs: Any*)

38.6.2 CheckDependenciesMeta

class flash.core.model.**CheckDependenciesMeta**(**args*, ***kwargs*)

38.6.3 ModuleWrapperBase

class flash.core.model.**ModuleWrapperBase**

The ModuleWrapperBase is a base for classes which wrap a LightningModule or an instance of ModuleWrapperBase.

This class ensures that trainer attributes are forwarded to any wrapped or nested LightningModule instances so that nested calls to .log are handled correctly. The ModuleWrapperBase is also stateful, meaning that a *DataPipelineState* can be attached. Attached state will be forwarded to any nested ModuleWrapperBase instances.

38.6.4 DatasetProcessor

class flash.core.model.**DatasetProcessor**

The `DatasetProcessor` mixin provides hooks for classes which need custom logic for producing the data loaders for each running stage given the corresponding dataset.

38.7 flash.core.registry

FlashRegistry	This class is used to register function or `functools.partial` class to a registry.
ExternalRegistry	The `ExternalRegistry` is a `FlashRegistry` that can point to an external provider via a getter function.
ConcatRegistry	The `ConcatRegistry` can be used to concatenate multiple registries of different types together.

38.7.1 FlashRegistry

class flash.core.registry.**FlashRegistry**(*name*, *verbose=False*)

This class is used to register function or `functools.partial` class to a registry.

get(*key*, *with_metadata=False*, *strict=True*, ***metadata*)

This function is used to gather matches from the registry:

Parameters

- **key** (`str`) – Name of the registered function.
- **with_metadata** (`bool`) – Whether to include the associated metadata in the return value.
- **strict** (`bool`) – Whether to return all matches or just one.
- **metadata** – Metadata used to filter against existing registry item's metadata.

Return type `Union[Callable, Dict[str, Any], List[Dict[str, Any]], List[Callable]]`

38.7.2 ExternalRegistry

class flash.core.registry.**ExternalRegistry**(*getter*, *name*, *providers=None*, *verbose=False*)

The `ExternalRegistry` is a `FlashRegistry` that can point to an external provider via a getter function.

Parameters

- **getter** (`Callable`) – A function whose first argument is a key that can optionally take additional args and kwargs.
- **providers** (`Union[Provider, List[Provider], None]`) – The provider(/s) of entries in this registry.

available_keys()

Since we don't know the available keys, just give a generic message.

Return type `List[str]`

get(*key*, *with_metadata=False*, *strict=True*, ***metadata*)

Returns a partial of the getter with the first argument as the given key and wrapped to print the providers.

Return type Union[Callable, Dict[str, Any], List[Dict[str, Any]], List[Callable]]

38.7.3 ConcatRegistry

class flash.core.registry.**ConcatRegistry**(**registries*)

The ConcatRegistry can be used to concatenate multiple registries of different types together.

38.8 flash.core.optimizers

LARS	Extends SGD in PyTorch with LARS scaling from the paper Large batch training of Convolutional Networks.
LAMB	Extends ADAM in pytorch to incorporate LAMB algorithm from the paper: Large batch optimization for deep learning: Training BERT in 76 minutes.
LinearWarmupCosineAnnealingLR	Sets the learning rate of each parameter group to follow a linear warmup schedule between warmup_start_lr and base_lr followed by a cosine annealing schedule between base_lr and eta_min.

38.8.1 LARS

class flash.core.optimizers.**LARS**(*params*, *lr=torch.optim.optimizer.required*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *trust_coefficient=0.001*, *eps=1e-08*)

Extends SGD in PyTorch with LARS scaling from the paper Large batch training of Convolutional Networks.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **trust_coefficient** (*float, optional*) – trust coefficient for computing LR (default: 0.001)
- **eps** (*float, optional*) – eps for division denominator (default: 1e-8)

Example

```
>>> model = nn.Linear(10, 1)
>>> optimizer = LARS(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> # loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Note: The application of momentum in the SGD part is modified according to the PyTorch standards. LARS scaling fits into the equation in the following fashion.

$$
\begin{aligned}
g_{t+1} &= \text{lars\_lr} * (\beta * p_t + g_{t+1}), \\
v_{t+1} &= \mu * v_t + g_{t+1}, \\
p_{t+1} &= p_t - \text{lr} * v_{t+1},
\end{aligned}
$$

where p, g, v, μ and β denote the parameters, gradient, velocity, momentum, and weight decay respectively. The $lars_lr$ is defined by Eq. 6 in the paper. The Nesterov version is analogously modified.

Warning: Parameters with weight decay set to 0 will automatically be excluded from layer-wise LR scaling. This is to ensure consistency with papers like SimCLR and BYOL.

step(*closure=None*)

Performs a single optimization step.

Parameters **closure**¶ (*callable, optional*) – A closure that reevaluates the model and returns the loss.

38.8.2 LAMB

class flash.core.optimizers.**LAMB**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-06*, *weight_decay=0*, *exclude_from_layer_adaptation=False*, *amsgrad=False*)

Extends ADAM in pytorch to incorporate LAMB algorithm from the paper: Large batch optimization for deep learning: Training BERT in 76 minutes.

Parameters

- **params**¶ (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr**¶ (*float*) – learning rate
- **betas**¶ (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps**¶ (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay**¶ (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **exclude_from_layer_adaptation**¶ (*bool, optional*) – layers which do not need LAMB layer adaptation (default: False)
- **amsgrad**¶ (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)

Example

```
>>> model = nn.Linear(10, 1)
>>> optimizer = LAMB(model.parameters(), lr=0.1)
>>> optimizer.zero_grad()
>>> # loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Warning: Since the default weight decay for LAMB is set to 0., we do not club together 0. weight decay and exclusion from layer adaptation like LARS. This would cause the optimizer to exclude all layers from layer adaptation.

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

38.8.3 LinearWarmupCosineAnnealingLR

class flash.core.optimizers.**LinearWarmupCosineAnnealingLR**(*optimizer*, *warmup_epochs*, *max_epochs*, *warmup_start_lr=0.0*, *eta_min=0.0*, *last_epoch=- 1*)

Sets the learning rate of each parameter group to follow a linear warmup schedule between warmup_start_lr and base_lr followed by a cosine annealing schedule between base_lr and eta_min.

Warning: It is recommended to call `step()` for *LinearWarmupCosineAnnealingLR* after each iteration as calling it after each epoch will keep the starting lr at warmup_start_lr for the first epoch which is 0 in most cases.

Warning: passing epoch to `step()` is being deprecated and comes with an EPOCH_DEPRECATION_WARNING. It calls the `_get_closed_form_lr()` method for this scheduler instead of *get_lr()*. Though this does not change the behavior of the scheduler, when passing epoch param to `step()`, the user should call the `step()` function before calling train and validation methods.

Example

```
>>> layer = nn.Linear(10, 1)
>>> optimizer = Adam(layer.parameters(), lr=0.02)
>>> scheduler = LinearWarmupCosineAnnealingLR(optimizer, warmup_epochs=10, max_
↪epochs=40)
>>> #
>>> # the default case
>>> for epoch in range(40):
...     # train(...)
...     # validate(...)
```

(continues on next page)

(continued from previous page)

```
...     scheduler.step()
>>> #
>>> # passing epoch param case
>>> for epoch in range(40):
...     scheduler.step(epoch)
...     # train(...)
...     # validate(...)
```

get_lr()
Compute learning rate using chainable form of the scheduler.

Return type List[float]

38.9 Utilities

from_argparse_args	Modified version of pytorch_lightning.utilities.argparse.from_argparse_args() which populates valid_kwargs from pytorch_lightning.Trainer.
get_callable_name	**rtype** str
get_callable_dict	**rtype** Union[Dict, Mapping]
predict_context	This decorator is used as context manager to put model in eval mode before running predict and reset to train after.

38.9.1 flash.core.trainer.from_argparse_args

flash.core.trainer.**from_argparse_args**(*cls*, *args*, ***kwargs*)
Modified version of pytorch_lightning.utilities.argparse.from_argparse_args() which populates valid_kwargs from pytorch_lightning.Trainer.

38.9.2 flash.core.utilities.apply_func.get_callable_name

flash.core.utilities.apply_func.**get_callable_name**(*fn_or_class*)

Return type str

38.9.3 flash.core.utilities.apply_func.get_callable_dict

flash.core.utilities.apply_func.**get_callable_dict**(*fn*)

Return type Union[Dict, Mapping]

38.9.4 flash.core.model.predict_context

flash.core.model.**predict_context**(*func*)

This decorator is used as context manager to put model in eval mode before running predict and reset to train after.

Return type Callable

CHAPTER

THIRTYNINE

FLASH.CORE.DATA

39.1 flash.core.data.auto_dataset

AutoDataset	The AutoDataset is a BaseAutoDataset and a Dataset.
BaseAutoDataset	The BaseAutoDataset class wraps the output of a call to *load_data()* and a DataSource and provides the _call_load_sample method to call *load_sample()* with the correct *CurrentRunningStageFuncContext* for the current running_stage.
IterableAutoDataset	The IterableAutoDataset is a BaseAutoDataset and a IterableDataset.

39.1.1 AutoDataset

class flash.core.data.auto_dataset.**AutoDataset**(*data*, *data_source*, *running_stage*)
The AutoDataset is a BaseAutoDataset and a Dataset.

The *data* argument must be a Sequence (it must have a length).

39.1.2 BaseAutoDataset

class flash.core.data.auto_dataset.**BaseAutoDataset**(*data*, *data_source*, *running_stage*)
The BaseAutoDataset class wraps the output of a call to *load_data()* and a DataSource and provides the _call_load_sample method to call *load_sample()* with the correct *CurrentRunningStageFuncContext* for the current running_stage. Inheriting classes are responsible for extracting samples from data to be given to _call_load_sample.

Parameters

- **data** (*~DATA_TYPE*) – The output of a call to *load_data()*.
- **data_source** (*DataSource*) – The *DataSource* which has the load_sample method.
- **running_stage** (RunningStage) – The current running stage.

39.1.3 IterableAutoDataset

class flash.core.data.auto_dataset.**IterableAutoDataset**(*data*, *data_source*, *running_stage*)
The IterableAutoDataset is a BaseAutoDataset and a IterableDataset.

The *data* argument must be an Iterable.

39.2 flash.core.data.base_viz

BaseVisualization	This Base Class is used to create visualization tool on top of *Preprocess* hooks.

39.2.1 BaseVisualization

class flash.core.data.base_viz.**BaseVisualization**(*enabled=False*)
This Base Class is used to create visualization tool on top of *Preprocess* hooks.

Override any of the show_{preprocess_hook_name} to receive the associated data and visualize them.

Example:

```
from flash.image import ImageClassificationData
from flash.core.data.base_viz import BaseVisualization

class CustomBaseVisualization(BaseVisualization):

    def show_load_sample(self, samples: List[Any], running_stage):
        # plot samples
```

(continues on next page)

(continued from previous page)

```
    def show_pre_tensor_transform(self, samples: List[Any], running_stage):
        # plot samples

    def show_to_tensor_transform(self, samples: List[Any], running_stage):
        # plot samples

    def show_post_tensor_transform(self, samples: List[Any], running_stage):
        # plot samples

    def show_collate(self, batch: List[Any], running_stage):
        # plot batch

    def show_per_batch_transform(self, batch: List[Any], running_stage):
        # plot batch

class CustomImageClassificationData(ImageClassificationData):

    @staticmethod
    def configure_data_fetcher(*args, **kwargs) -> BaseDataFetcher:
        return CustomBaseVisualization(*args, **kwargs)

dm = CustomImageClassificationData.from_folders(
    train_folder="./data/train",
    val_folder="./data/val",
    test_folder="./data/test",
    predict_folder="./data/predict")

# visualize a ``train`` batch
dm.show_train_batches()

# visualize next ``train`` batch
dm.show_train_batches()

# visualize a ``val`` batch
dm.show_val_batches()

# visualize a ``test`` batch
dm.show_test_batches()

# visualize a ``predict`` batch
dm.show_predict_batches()
```

Note: If the user wants to plot all different transformation stages at once, override the `show` function directly.

Example:

```
class CustomBaseVisualization(BaseVisualization):

    def show(self, batch: Dict[str, Any], running_stage: RunningStage):
        print(batch)
        # out
```

(continues on next page)

(continued from previous page)

```
        {
            'load_sample': [...],
            'pre_tensor_transform': [...],
            'to_tensor_transform': [...],
            'post_tensor_transform': [...],
            'collate': [...],
            'per_batch_transform': [...],
        }
```

Note: As the *Preprocess* hooks are injected within the threaded workers of the DataLoader, the data won't be accessible when using `num_workers > 0`.

show(*batch*, *running_stage*, *func_names_list*)
Override this function when you want to visualize a composition.

Return type None

show_collate(*batch*, *running_stage*)
Override to visualize preprocess `collate` output data.

Return type None

show_load_sample(*samples*, *running_stage*)
Override to visualize preprocess `load_sample` output data.

show_per_batch_transform(*batch*, *running_stage*)
Override to visualize preprocess `per_batch_transform` output data.

Return type None

show_per_batch_transform_on_device(*batch*, *running_stage*)
Override to visualize preprocess `per_batch_transform_on_device` output data.

Return type None

show_per_sample_transform_on_device(*samples*, *running_stage*)
Override to visualize preprocess `per_sample_transform_on_device` output data.

Return type None

show_post_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `post_tensor_transform` output data.

show_pre_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `pre_tensor_transform` output data.

show_to_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `to_tensor_transform` output data.

39.3 flash.core.data.batch

default_uncollate	This function is used to uncollate a batch into samples.

39.3.1 flash.core.data.batch.default_uncollate

flash.core.data.batch.**default_uncollate**(*batch*)

This function is used to uncollate a batch into samples. .. rubric:: Examples

```
>>> a, b = default_uncollate(torch.rand((2,1)))
```

39.4 flash.core.data.callback

BaseDataFetcher	This class is used to profile *Preprocess* hook outputs.
ControlFlow	
FlashCallback	FlashCallback is an extension of pytorch_lightning.callbacks.Callback.

39.4.1 BaseDataFetcher

class flash.core.data.callback.**BaseDataFetcher**(*enabled=False*)

This class is used to profile *Preprocess* hook outputs.

By default, the callback won't profile the data being processed as it may lead to OOMError.

Example:

```
from flash.core.data.callback import BaseDataFetcher
from flash.core.data.data_module import DataModule
from flash.core.data.data_source import DataSource
from flash.core.data.process import Preprocess

class CustomPreprocess(Preprocess):

    def __init__(**kwargs):
        super().__init__(
            data_sources = {"inputs": DataSource()},
            **kwargs,
        )

class PrintData(BaseDataFetcher):

    def print(self):
        print(self.batches)

class CustomDataModule(DataModule):
```

(continues on next page)

(continued from previous page)

```
    preprocess_cls = CustomPreprocess

    @staticmethod
    def configure_data_fetcher():
        return PrintData()

    @classmethod
    def from_inputs(
        cls,
        train_data: Any,
        val_data: Any,
        test_data: Any,
        predict_data: Any,
    ) -> "CustomDataModule":
        return cls.from_data_source(
            "inputs",
            train_data=train_data,
            val_data=val_data,
            test_data=test_data,
            predict_data=predict_data,
            batch_size=5,
        )

dm = CustomDataModule.from_inputs(range(5), range(5), range(5), range(5))
data_fetcher = dm.data_fetcher

# By default, the ``data_fetcher`` is disabled to prevent OOM.
# The ``enable`` context manager will activate it.
with data_fetcher.enable():

    # This will fetch the first val dataloader batch.
    _ = next(iter(dm.val_dataloader()))

data_fetcher.print()
# out:
{
    'train': {},
    'test': {},
    'val': {
        'load_sample': [0, 1, 2, 3, 4],
        'pre_tensor_transform': [0, 1, 2, 3, 4],
        'to_tensor_transform': [0, 1, 2, 3, 4],
        'post_tensor_transform': [0, 1, 2, 3, 4],
        'collate': [tensor([0, 1, 2, 3, 4])],
        'per_batch_transform': [tensor([0, 1, 2, 3, 4])]},
    'predict': {}
}
data_fetcher.reset()
data_fetcher.print()
# out:
{
    'train': {},
```

(continues on next page)

(continued from previous page)

```
    'test': {},
    'val': {},
    'predict': {}
}
```

enable()
This function is used to enable to BaseDataFetcher.

39.4.2 ControlFlow

class flash.core.data.callback.**ControlFlow**(*callbacks*)

39.5 flash.core.data.data_module

DataModule	A basic DataModule class for all Flash tasks.

39.6 flash.core.data.data_pipeline

DataPipeline	DataPipeline holds the engineering logic to connect *Preprocess* and/or *Postprocess* objects to the DataModule, Flash Task and Trainer.
DataPipelineState	A class to store and share all process states once a *DataPipeline* has been initialized.

39.6.1 DataPipeline

class flash.core.data.data_pipeline.**DataPipeline**(*data_source=None*, *preprocess=None*, *postprocess=None*, *deserializer=None*, *serializer=None*)

DataPipeline holds the engineering logic to connect *Preprocess* and/or *Postprocess* objects to the DataModule, Flash Task and Trainer.

Example:

```
class CustomPreprocess(Preprocess):
    pass

class CustomPostprocess(Postprocess):
    pass

custom_data_pipeline = DataPipeline(CustomPreprocess(), CustomPostprocess())

# And it can attached to both the datamodule and model.

datamodule.data_pipeline = custom_data_pipeline
model.data_pipeline = custom_data_pipeline
```

initialize(*data_pipeline_state=None*)
Creates the *DataPipelineState* and gives the reference to the: *Preprocess*, *Postprocess*, and *Serializer*. Once this has been called, any attempt to add new state will give a warning.

Return type *DataPipelineState*

39.6.2 DataPipelineState

class flash.core.data.data_pipeline.**DataPipelineState**
A class to store and share all process states once a *DataPipeline* has been initialized.

get_state(*state_type*)
Get the *ProcessState* of the given type from the *DataPipelineState*.

Return type Optional[*ProcessState*]

set_state(*state*)
Add the given *ProcessState* to the *DataPipelineState*.

39.7 flash.core.data.data_source

DatasetDataSource	The `DatasetDataSource` implements default behaviours for data sources which expect the input to *load_data()* to be a `torch.utils.data.dataset.Dataset`
DataSource	The `DataSource` class encapsulates two hooks: `load_data` and `load_sample`.
DefaultDataKeys	The `DefaultDataKeys` enum contains the keys that are used by built-in data sources to refer to inputs and targets.
DefaultDataSources	The `DefaultDataSources` enum contains the data source names used by all of the default `from_*` methods in *DataModule*.
FiftyOneDataSource	The `FiftyOneDataSource` expects the input to *load_data()* to be a `fiftyone.core.collections.SampleCollection`.
ImageLabelsMap	
LabelsState	A *ProcessState* containing `labels`, a mapping from class index to label.
MockDataset	The `MockDataset` catches any metadata that is attached through `__setattr__`.
NumpyDataSource	The `NumpyDataSource` is a `SequenceDataSource` which expects the input to *load_data()* to be a sequence of `np.ndarray` objects.
PathsDataSource	The `PathsDataSource` implements default behaviours for data sources which expect the input to *load_data()* to be either a directory with a subdirectory for each class or a tuple containing list of files and corresponding list of targets.

continues on next page

Table 7 – continued from previous page

SequenceDataSource	The SequenceDataSource implements default behaviours for data sources which expect the input to *load_data()* to be a sequence of tuples ((input, target) where target can be None).
TensorDataSource	The TensorDataSource is a SequenceDataSource which expects the input to *load_data()* to be a sequence of torch.Tensor objects.

39.7.1 DatasetDataSource

class flash.core.data.data_source.**DatasetDataSource**

The DatasetDataSource implements default behaviours for data sources which expect the input to *load_data()* to be a torch.utils.data.dataset.Dataset

Parameters **labels**¶ – Optionally pass the labels as a mapping from class index to label string. These will then be set as the *LabelsState*.

39.7.2 DefaultDataKeys

class flash.core.data.data_source.**DefaultDataKeys**(**args*, ***kwargs*)

The DefaultDataKeys enum contains the keys that are used by built-in data sources to refer to inputs and targets.

39.7.3 DefaultDataSources

class flash.core.data.data_source.**DefaultDataSources**(**args*, ***kwargs*)

The DefaultDataSources enum contains the data source names used by all of the default from_* methods in *DataModule*.

39.7.4 FiftyOneDataSource

class flash.core.data.data_source.**FiftyOneDataSource**(*label_field='ground_truth'*)

The FiftyOneDataSource expects the input to *load_data()* to be a fiftyone.core.collections.SampleCollection.

39.7.5 ImageLabelsMap

class flash.core.data.data_source.**ImageLabelsMap**(*labels_map*)

39.7.6 LabelsState

class flash.core.data.data_source.**LabelsState**(*labels*)
A *ProcessState* containing `labels`, a mapping from class index to label.

39.7.7 MockDataset

class flash.core.data.data_source.**MockDataset**
The `MockDataset` catches any metadata that is attached through `__setattr__`.

This is passed to *load_data()* so that attributes can be set on the generated data set.

39.7.8 NumpyDataSource

class flash.core.data.data_source.**NumpyDataSource**(*labels=None*)
The `NumpyDataSource` is a `SequenceDataSource` which expects the input to *load_data()* to be a sequence of `np.ndarray` objects.

39.7.9 PathsDataSource

class flash.core.data.data_source.**PathsDataSource**(*extensions=None*, *loader=None*, *labels=None*)
The `PathsDataSource` implements default behaviours for data sources which expect the input to *load_data()* to be either a directory with a subdirectory for each class or a tuple containing list of files and corresponding list of targets.

Parameters

- **extensions** (Optional[Tuple[str, ...]]) – The file extensions supported by this data source (e.g. `(".jpg", ".png")`).
- **labels** (Optional[Sequence[str]]) – Optionally pass the labels as a mapping from class index to label string. These will then be set as the *LabelsState*.

static find_classes(*dir*)
Finds the class folders in a dataset. Ensures that no class is a subdirectory of another.

Parameters **dir** (str) – Root directory path.

Returns (classes, class_to_idx) where classes are relative to (dir), and class_to_idx is a dictionary.

Return type tuple

39.7.10 SequenceDataSource

class flash.core.data.data_source.**SequenceDataSource**(*labels=None*)
The `SequenceDataSource` implements default behaviours for data sources which expect the input to *load_data()* to be a sequence of tuples (`(input, target)` where target can be `None`).

Parameters **labels** (Optional[Sequence[str]]) – Optionally pass the labels as a mapping from class index to label string. These will then be set as the *LabelsState*.

39.7.11 TensorDataSource

class flash.core.data.data_source.**TensorDataSource**(*labels=None*)

The TensorDataSource is a SequenceDataSource which expects the input to *load_data()* to be a sequence of torch.Tensor objects.

has_file_allowed_extension	Checks if a file is an allowed extension.
has_len	**rtype** bool
make_dataset	Generates a list of samples of a form (path_to_sample, class).

39.7.12 flash.core.data.data_source.has_file_allowed_extension

flash.core.data.data_source.**has_file_allowed_extension**(*filename*, *extensions*)

Checks if a file is an allowed extension.

Parameters

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

Returns True if the filename ends with one of given extensions

Return type bool

39.7.13 flash.core.data.data_source.has_len

flash.core.data.data_source.**has_len**(*data*)

Return type bool

39.7.14 flash.core.data.data_source.make_dataset

flash.core.data.data_source.**make_dataset**(*directory*, *class_to_idx*, *extensions=None*, *is_valid_file=None*)

Generates a list of samples of a form (path_to_sample, class).

Parameters

- **directory** (*str*) – root dataset directory
- **class_to_idx** (*Dict[str, int]*) – dictionary mapping class name to class index
- **extensions** (*optional*) – A list of allowed extensions. Either extensions or is_valid_file should be passed. Defaults to None.
- **is_valid_file** (*optional*) – A function that takes path of a file and checks if the file is a valid file (used to check of corrupt files) both extensions and is_valid_file should not be passed. Defaults to None.

Raises ValueError – In case extensions and is_valid_file are None or both are not None.

Returns samples of a form (path_to_sample, class)

Return type List[Tuple[str, int]]

39.8 flash.core.data.process

BasePreprocess	
DefaultPreprocess	
DeserializerMapping	Deserializer Mapping.
Deserializer	Deserializer.
Postprocess	The *Postprocess* encapsulates all the data processing logic that should run after the model.
Preprocess	The *Preprocess* encapsulates all the data processing logic that should run before the data is passed to the model.
SerializerMapping	If the model output is a dictionary, then the *SerializerMapping* enables each entry in the dictionary to be passed to it's own *Serializer*.
Serializer	A *Serializer* encapsulates a single `serialize` method which is used to convert the model output into the desired output format when predicting.

39.8.1 BasePreprocess

class flash.core.data.process.**BasePreprocess**

abstract get_state_dict()
Override this method to return state_dict.

Return type Dict[str, Any]

abstract classmethod load_state_dict(*state_dict*, *strict=False*)
Override this method to load from state_dict.

39.8.2 DefaultPreprocess

class flash.core.data.process.**DefaultPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_sources=None*, *default_data_source=None*)

39.8.3 DeserializerMapping

class flash.core.data.process.**DeserializerMapping**(*deserializers*)

Deserializer Mapping.

39.8.4 Deserializer

class flash.core.data.process.**Deserializer**

Deserializer.

39.8.5 SerializerMapping

class flash.core.data.process.**SerializerMapping**(*serializers*)

If the model output is a dictionary, then the *SerializerMapping* enables each entry in the dictionary to be passed to it's own *Serializer*.

39.9 flash.core.data.properties

ProcessState	Base class for all process states.
Properties	

39.9.1 ProcessState

class flash.core.data.properties.**ProcessState**

Base class for all process states.

39.9.2 Properties

class flash.core.data.properties.**Properties**

39.10 flash.core.data.splits

SplitDataset	SplitDataset is used to create Dataset Subset using indices.

39.10.1 SplitDataset

class flash.core.data.splits.**SplitDataset**(*dataset*, *indices=None*, *use_duplicated_indices=False*)

SplitDataset is used to create Dataset Subset using indices.

Parameters

- **dataset** (Any) – A dataset to be splitted
- **indices** (Optional[List[int]]) – List of indices to expose from the dataset
- **use_duplicated_indices** (bool) – Whether to allow duplicated indices.

Example:

```
split_ds = SplitDataset(dataset, indices=[10, 14, 25])

split_ds = SplitDataset(dataset, indices=[10, 10, 10, 14, 25], use_duplicated_
↪indices=True)
```

39.11 flash.core.data.transforms

ApplyToKeys	The `ApplyToKeys` class is an `nn.Sequential` which applies the given transforms to the given keys from the input.
KorniaParallelTransforms	The `KorniaParallelTransforms` class is an `nn.Sequential` which will apply the given transforms to each input (to `.forward`) in parallel, whilst sharing the random state (`._params`).

39.11.1 ApplyToKeys

class flash.core.data.transforms.**ApplyToKeys**(*keys*, **args*)

The `ApplyToKeys` class is an `nn.Sequential` which applies the given transforms to the given keys from the input. When a single key is given, a single value will be passed to the transforms. When multiple keys are given, the corresponding values will be passed to the transforms as a list.

Parameters

- **keys** (Union[str, Sequence[str]]) – The key (`str`) or sequence of keys

(Sequence[str]) to extract and forward to the transforms.

- **args** – The transforms, passed to the nn.Sequential super constructor.

39.11.2 KorniaParallelTransforms

class flash.core.data.transforms.**KorniaParallelTransforms**(**args: Any*, ***kwargs: Any*)

The KorniaParallelTransforms class is an nn.Sequential which will apply the given transforms to each input (to .forward) in parallel, whilst sharing the random state (._params). This should be used when multiple elements need to be augmented in the same way (e.g. an image and corresponding segmentation mask).

Parameters **args** – The transforms, passed to the nn.Sequential super constructor.

merge_transforms	Utility function to merge two transform dictionaries.
kornia_collate	Kornia transforms add batch dimension which need to be removed.

39.11.3 flash.core.data.transforms.merge_transforms

flash.core.data.transforms.**merge_transforms**(*base_transforms*, *additional_transforms*)

Utility function to merge two transform dictionaries. For each hook, the additional_transforms will be be called after the base_transforms.

Parameters

- **base_transforms** (Dict[str, Callable]) – The base transforms dictionary.
- **additional_transforms** (Dict[str, Callable]) – The dictionary of additional transforms to be appended to the base_transforms.

Return type Dict[str, Callable]

Returns The new dictionary of transforms.

39.11.4 flash.core.data.transforms.kornia_collate

flash.core.data.transforms.**kornia_collate**(*samples*)

Kornia transforms add batch dimension which need to be removed.

This function removes that dimension and then applies torch.utils.data._utils.collate.default_collate.

Return type Dict[str, Any]

39.12 flash.core.data.utils

CurrentFuncContext	
CurrentRunningStageContext	

continues on next page

Table 14 – continued from previous page

CurrentRunningStageFuncContext	
FuncModule	This class is used to wrap a callable within a nn.Module and apply the wrapped function in *__call__*

39.12.1 CurrentFuncContext

class flash.core.data.utils.**CurrentFuncContext**(*current_fn*, *obj*)

39.12.2 CurrentRunningStageContext

class flash.core.data.utils.**CurrentRunningStageContext**(*running_stage*, *obj*, *reset=True*)

39.12.3 CurrentRunningStageFuncContext

class flash.core.data.utils.**CurrentRunningStageFuncContext**(*running_stage*, *current_fn*, *obj*)

39.12.4 FuncModule

class flash.core.data.utils.**FuncModule**(*func*)

This class is used to wrap a callable within a nn.Module and apply the wrapped function in *__call__*

convert_to_modules	
download_data	Download file with progressbar.

39.12.5 flash.core.data.utils.convert_to_modules

flash.core.data.utils.**convert_to_modules**(*transforms*)

39.12.6 flash.core.data.utils.download_data

flash.core.data.utils.**download_data**(*url*, *path='data/'*, *verbose=False*)

Download file with progressbar.

Code taken from: https://gist.github.com/ruxi/5d6803c116ec1130d484a4ab8c00c603 # __author__ = "github.com/ruxi" # __license__ = "MIT"

Usage: download_file('http://web4host.net/5MB.zip')

Return type None

CHAPTER

FORTY

FLASH.CORE.SERVE

ModelComponent	alias of `object`
Composition	Create a composition which define computations / endpoints to create & run.
Endpoint	An endpoint maps a route and request/response payload to components.
Servable	ModuleWrapperBase around a model object to enable serving at scale.
expose	Expose a function/method via a web API for serving model inference.

40.1 flash.core.serve.component.ModelComponent

`flash.core.serve.component.`**`ModelComponent`**
alias of `object`

40.2 flash.core.serve.composition.Composition

class `flash.core.serve.composition.`**`Composition`**(*, *DEBUG=False*, *TESTING=False*, ***kwargs*)
Create a composition which define computations / endpoints to create & run.

Any number of components are accepted, which may have aribtrary connections between them. The final path through the component/connection DAG is determined by the root/terminal node position as specified by endpoint input/outputs keys.

If only ONE component is provided, there is no need to create an Endpoint object. The library will generate a fully connected input/ouput endpoint for the one component with the *route* name set by the name of the method the *@expose* decorator is applied to.

Parameters **kwargs**¶ (`Union`[`object`, *Endpoint*]) – Assignment of human readable names to `ModelComponent` and `Endpoint` instances. If more than one `ModelComponent` is passed, an `Endpoint` is needed as well.

Warning:

- This is a Work In Progress interface!

40.3 flash.core.serve.core.Endpoint

class flash.core.serve.core.**Endpoint**(*route*, *inputs*, *outputs*)

An endpoint maps a route and request/response payload to components.

Parameters

- **route** (*str*) – The API route name to construct as the servicing POST endpoint.
- **inputs** (*Dict[str, str]*) – The full name of a component input. Typically specified by just passing in the component parameter attribute (ie.``component.inputs.foo``).
- **outputs** (*Dict[str, str]*) – The full name of a component output. Typically specified by just passing in the component parameter attribute (ie.``component.outputs.bar``).

40.4 flash.core.serve.core.Servable

class flash.core.serve.core.**Servable**(**args*, *download_path=None*, *script_loader_cls=<class 'flash.core.serve.core.FlashServeScriptLoader'>*)

ModuleWrapperBase around a model object to enable serving at scale.

Create a `Servable` from either (LM, LOCATION) or (LOCATION,)

Parameters

- ***args** – A model class and path to the asset file (url or local file path) OR a singular path to a torchscript asset which can be loaded without the model class definition.
- **download_path** (`Optional`[`Path`]) – Optional url to download a model from.

40.5 flash.core.serve.decorators.expose

flash.core.serve.decorators.**expose**(*inputs*, *outputs*)

Expose a function/method via a web API for serving model inference.

The `@expose` decorator has two arguments, inputs and outputs, which describe how the inputs to predict are decoded from the request and how the outputs of predict are encoded to a response.

Must decorate one (and only one) method when used within a subclass of `ModelComponent`.

Parameters

- **inputs** (`Dict`[`str`, `BaseType`]) – accepts a dictionary mapping keys to decorated method parameter names (must be one to one mapping) with values corresponding to an instantiated specification of a Flash Serve Data Type (ie. `Number()`, `Image()`, `Text()`, etc...)
- **outputs** (`Dict`[`str`, `BaseType`]) – accepts a dictionary mapping outputs of the decorated method to keys and data type (similar to inputs). However, unlike `inputs` the output keys are less strict in their names. IF the method returns a dictionary, the keys must match one-to-one. However, if the method returns a sorted sequence (list / tuple) the keys can be arbitrary, so long as no reserved names are used (primarily python keywords). For result sequences, the order in which keys are defined maps to the appropriate element index in the result (ie. `key 0 -> sequence[0]`, `key 1 -> sequence[1]`, etc.)

CHAPTER

FORTYONE

FLASH.IMAGE

41.1 Classification

`ImageClassifier`	The `ImageClassifier` is a `Task` for classifying images.
`ImageClassificationData`	Data module for image classification tasks.
`ImageClassificationPreprocess`	
`classification.data.MatplotlibVisualization`	Process and show the image batch and its associated label using matplotlib.

41.1.1 ImageClassifier

class flash.image.classification.model.**ImageClassifier**(*num_classes*, *backbone='resnet18'*, *backbone_kwargs=None*, *head=None*, *pretrained=True*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.001*, *multi_label=False*, *serializer=None*)

The `ImageClassifier` is a `Task` for classifying images. For more details, see *Image Classification*. The `ImageClassifier` also supports multi-label classification with `multi_label=True`. For more details, see *Multi-label Image Classification*.

You can register custom backbones to use with the `ImageClassifier`:

```
from torch import nn
import torchvision
from flash.image import ImageClassifier

# This is useful to create new backbone and make them accessible from␣
↪`ImageClassifier`
@ImageClassifier.backbones(name="resnet18")
def fn_resnet(pretrained: bool = True):
    model = torchvision.models.resnet18(pretrained)
    # remove the last two layers & turn it into a Sequential model
    backbone = nn.Sequential(*list(model.children())[:-2])
    num_features = model.fc.in_features
    # backbones need to return the num_features to build the head
    return backbone, num_features
```

Parameters

- **num_classes** (int) – Number of classes to classify.
- **backbone** (Union[str, Tuple[Module, int]]) – A string or (model, num_features) tuple to use to compute image features, defaults to "resnet18".
- **pretrained** (Union[bool, str]) – A bool or string to specify the pretrained weights of the backbone, defaults to True which loads the default supervised pretrained weights.
- **loss_fn** (Optional[Callable]) – Loss function for training, defaults to torch.nn.functional.cross_entropy().
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training, defaults to torch.optim.SGD.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inheriting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to torchmetrics.Accuracy.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

41.1.2 ImageClassificationData

class flash.image.classification.data.**ImageClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for image classification tasks.

classmethod from_csv(*input_field*, *target_fields=None*, *train_file=None*, *train_images_root=None*, *train_resolver=None*, *val_file=None*, *val_images_root=None*, *val_resolver=None*, *test_file=None*, *test_images_root=None*, *test_resolver=None*, *predict_file=None*, *predict_images_root=None*, *predict_resolver=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *ImageClassificationData* object from the given CSV files using the *DataSource* of name `CSV` from the passed or constructed *Preprocess*.

Parameters

- **input_field** (`str`) – The field (column) in the CSV file to use for the input.
- **target_fields** (`Union[str, Sequence[str], None]`) – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (`Optional[str]`) – The CSV file containing the training data.
- **train_images_root** (`Optional[str]`) – The directory containing the train images. If `None`, the directory containing the `train_file` will be used.
- **train_resolver** (`Optional[Callable[[str, str], str]]`) – The function to use to resolve filenames given the `train_images_root` and IDs from the `input_field` column.
- **val_file** (`Optional[str]`) – The CSV file containing the validation data.
- **val_images_root** (`Optional[str]`) – The directory containing the validation images. If `None`, the directory containing the `val_file` will be used.
- **val_resolver** (`Optional[Callable[[str, str], str]]`) – The function to use to resolve filenames given the `val_images_root` and IDs from the `input_field` column.
- **test_file** (`Optional[str]`) – The CSV file containing the testing data.
- **test_images_root** (`Optional[str]`) – The directory containing the test images. If `None`, the directory containing the `test_file` will be used.
- **test_resolver** (`Optional[Callable[[str, str], str]]`) – The function to use to resolve filenames given the `test_images_root` and IDs from the `input_field` column.
- **predict_file** (`Optional[str]`) – The CSV file containing the data to use when predicting.
- **predict_images_root** (`Optional[str]`) – The directory containing the predict images. If `None`, the directory containing the `predict_file` will be used.

- **predict_resolver** (Optional[Callable[[str, str], str]]) – The function to use to resolve filenames given the `predict_images_root` and IDs from the `input_field` column.
- **train_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The `Preprocess` to pass to the *DataModule*. If `None`, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The `sampler` to use for the `train_dataloader`.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

classmethod **from_data_frame**(*input_field*, *target_fields=None*, *train_data_frame=None*, *train_images_root=None*, *train_resolver=None*, *val_data_frame=None*, *val_images_root=None*, *val_resolver=None*, *test_data_frame=None*, *test_images_root=None*, *test_resolver=None*, *predict_data_frame=None*, *predict_images_root=None*, *predict_resolver=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *ImageClassificationData* object from the given pandas `DataFrame` objects.

Parameters

- **input_field** (str) – The field (column) in the pandas `DataFrame` to use for the input.
- **target_fields** (Union[str, Sequence[str], None]) – The field or fields (columns) in the pandas `DataFrame` to use for the target.
- **train_data_frame** (Optional[DataFrame]) – The pandas `DataFrame` containing the training data.

- **train_images_root** (Optional[str]) – The directory containing the train images. If None, values in the input_field will be assumed to be the full file paths.
- **train_resolver** (Optional[Callable[[str, str], str]]) – The function to use to resolve filenames given the train_images_root and IDs from the input_field column.
- **val_data_frame** (Optional[DataFrame]) – The pandas DataFrame containing the validation data.
- **val_images_root** (Optional[str]) – The directory containing the validation images. If None, the directory containing the val_file will be used.
- **val_resolver** (Optional[Callable[[str, str], str]]) – The function to use to resolve filenames given the val_images_root and IDs from the input_field column.
- **test_data_frame** (Optional[DataFrame]) – The pandas DataFrame containing the testing data.
- **test_images_root** (Optional[str]) – The directory containing the test images. If None, the directory containing the test_file will be used.
- **test_resolver** (Optional[Callable[[str, str], str]]) – The function to use to resolve filenames given the test_images_root and IDs from the input_field column.
- **predict_data_frame** (Optional[DataFrame]) – The pandas DataFrame containing the data to use when predicting.
- **predict_images_root** (Optional[str]) – The directory containing the predict images. If None, the directory containing the predict_file will be used.
- **predict_resolver** (Optional[Callable[[str, str], str]]) – The function to use to resolve filenames given the predict_images_root and IDs from the input_field column.
- **train_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Union[Callable, List, Dict[str, Callable], None]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.

- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

set_block_viz_window(*value*)
Setter method to switch on/off matplotlib to pop up windows.

Return type None

41.1.3 ImageClassificationPreprocess

class flash.image.classification.data.**ImageClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(196, 196)*, *deserializer=None*, ***data_source_kwargs*)

41.1.4 MatplotlibVisualization

class flash.image.classification.data.**MatplotlibVisualization**(*enabled=False*)
Process and show the image batch and its associated label using matplotlib.

classification.transforms.default_transforms	The default transforms for image classification: resize the image, convert the image and target to a tensor, collate the batch, and apply normalization.
classification.transforms.train_default_transforms	During training, we apply the default transforms with additional RandomHorizontalFlip.

41.1.5 flash.image.classification.transforms.default_transforms

flash.image.classification.transforms.**default_transforms**(*image_size*)
The default transforms for image classification: resize the image, convert the image and target to a tensor, collate the batch, and apply normalization.

Return type Dict[str, Callable]

41.1.6 flash.image.classification.transforms.train_default_transforms

flash.image.classification.transforms.**train_default_transforms**(*image_size*)

During training, we apply the default transforms with additional `RandomHorizontalFlip`.

Return type `Dict[str, Callable]`

41.2 Object Detection

ObjectDetector	The `ObjectDetector` is a `Task` for detecting objects in images.
ObjectDetectionData	
detection.data.FiftyOneParser	
detection.data.ObjectDetectionFiftyOneDataSource	
detection.data.ObjectDetectionPreprocess	
detection.serialization.FiftyOneDetectionLabels	A *Serializer* which converts model outputs to FiftyOne detection format.

41.2.1 ObjectDetector

class flash.image.detection.model.**ObjectDetector**(*num_classes*, *backbone='resnet18_fpn'*, *head='retinanet'*, *pretrained=True*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=0.005*, *serializer=None*, ***kwargs*)

The `ObjectDetector` is a `Task` for detecting objects in images. For more details, see *Object Detection*.

Parameters

- **num_classes** (`int`) – the number of classes for detection, including background
- **model** – a string of :attr`_models`. Defaults to 'fasterrcnn'.
- **backbone** (`Optional[str]`) – Pretrained backbone CNN architecture. Constructs a model with a ResNet-50-FPN backbone when no backbone is specified.
- **fpn** – If True, creates a Feature Pyramind Network on top of Resnet based CNNs.
- **pretrained** (`bool`) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** – number of trainable resnet layers starting from final block. Only applicable for *fasterrcnn*.
- **loss** – the function(s) to update the model with. Has no effect for torchvision detection models.
- **metrics** – The provided metrics. All metrics here will be logged to progress bar and the respective logger. Changing this argument currently has no effect.

- **optimizer** (Type[Optimizer]) – The optimizer to use for training. Can either be the actual class or the class name.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **pretrained** – Whether the model from torchvision should be loaded with it's pretrained weights. Has no effect for custom models.
- **learning_rate** (float) – The learning rate to use for training

41.2.2 ObjectDetectionData

class flash.image.detection.data.**ObjectDetectionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

classmethod **from_coco**(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *ObjectDetectionData* object from the given data folders and annotation files in the COCO format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.

- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Returns The constructed data module.

Examples:

```
data_module = ObjectDetectionData.from_coco(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

classmethod from_via(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *ObjectDetectionData* object from the given data folders and annotation files in the VIA format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.

- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = ObjectDetectionData.from_via(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

classmethod from_voc(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *ObjectDetectionData* object from the given data folders and annotation files in the VOC format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.

- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = ObjectDetectionData.from_voc(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

41.2.3 FiftyOneParser

class flash.image.detection.data.**FiftyOneParser**(*data*, *class_map*, *label_field*, *iscrowd*)

41.2.4 ObjectDetectionFiftyOneDataSource

class flash.image.detection.data.**ObjectDetectionFiftyOneDataSource**(*label_field='ground_truth'*, *iscrowd='iscrowd'*)

41.2.5 ObjectDetectionPreprocess

class flash.image.detection.data.**ObjectDetectionPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(128, 128)*, *parser=None*, ***data_source_kwargs*)

41.2.6 FiftyOneDetectionLabels

class flash.image.detection.serialization.**FiftyOneDetectionLabels**(*labels=None*, *threshold=None*, *return_filepath=False*)

A *Serializer* which converts model outputs to FiftyOne detection format.

Parameters

- **labels** (Optional[List[str]]) – A list of labels, assumed to map the class index to the label for that class. If labels is not provided, will attempt to get them from the *LabelsState*.
- **threshold** (Optional[float]) – a score threshold to apply to candidate detections.

- **return_filepath** (bool) – Boolean determining whether to return a dict containing filepath and FiftyOne labels (True) or only a list of FiftyOne labels (False)

41.3 Keypoint Detection

KeypointDetector	The ObjectDetector is a Task for detecting objects in images.
KeypointDetectionData	
keypoint_detection.data.KeypointDetectionPreprocess	

41.3.1 KeypointDetector

class flash.image.keypoint_detection.model.**KeypointDetector**(*num_keypoints*, *num_classes=2*, *backbone='resnet18_fpn'*, *head='keypoint_rcnn'*, *pretrained=True*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=0.0005*, *serializer=None*, ***kwargs*)

The ObjectDetector is a Task for detecting objects in images. For more details, see *Object Detection*.

Parameters

- **num_classes** (int) – the number of classes for detection, including background
- **model** – a string of :attr`_models`. Defaults to 'fasterrcnn'.
- **backbone** (Optional[str]) – Pretained backbone CNN architecture. Constructs a model with a ResNet-50-FPN backbone when no backbone is specified.
- **fpn** – If True, creates a Feature Pyramind Network on top of Resnet based CNNs.
- **pretrained** (bool) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** – number of trainable resnet layers starting from final block. Only applicable for *fasterrcnn*.
- **loss** – the function(s) to update the model with. Has no effect for torchvision detection models.
- **metrics** – The provided metrics. All metrics here will be logged to progress bar and the respective logger. Changing this argument currently has no effect.
- **optimizer** (Type[Optimizer]) – The optimizer to use for training. Can either be the actual class or the class name.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).

- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **pretrained** – Whether the model from torchvision should be loaded with it's pretrained weights. Has no effect for custom models.
- **learning_rate** (float) – The learning rate to use for training

41.3.2 KeypointDetectionData

class flash.image.keypoint_detection.data.**KeypointDetectionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

classmethod **from_coco**(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *KeypointDetectionData* object from the given data folders and annotation files in the COCO format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.

- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = KeypointDetectionData.from_coco(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

41.3.3 KeypointDetectionPreprocess

class flash.image.keypoint_detection.data.**KeypointDetectionPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(128, 128)*, *parser=None*)

41.4 Instance Segmentation

InstanceSegmentation	The InstanceSegmentation is a Task for detecting objects in images.
InstanceSegmentationData	
instance_segmentation.data.InstanceSegmentationPreprocess	

41.4.1 InstanceSegmentation

class flash.image.instance_segmentation.model.**InstanceSegmentation**(*num_classes*, *backbone='resnet18_fpn'*, *head='mask_rcnn'*, *pretrained=True*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=0.0005*, *serializer=None*, ***kwargs*)

The InstanceSegmentation is a Task for detecting objects in images. For more details, see *Object Detection*.

Parameters

- **num_classes** (int) – the number of classes for detection, including background
- **model** – a string of :attr`_models`. Defaults to 'fasterrcnn'.
- **backbone** (Optional[str]) – Pretained backbone CNN architecture. Constructs a model with a ResNet-50-FPN backbone when no backbone is specified.
- **fpn** – If True, creates a Feature Pyramind Network on top of Resnet based CNNs.
- **pretrained** (bool) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** – number of trainable resnet layers starting from final block. Only applicable for *fasterrcnn*.
- **loss** – the function(s) to update the model with. Has no effect for torchvision detection models.
- **metrics** – The provided metrics. All metrics here will be logged to progress bar and the respective logger. Changing this argument currently has no effect.
- **optimizer** (Type[Optimizer]) – The optimizer to use for training. Can either be the actual class or the class name.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **pretrained** – Whether the model from torchvision should be loaded with it's pretrained weights. Has no effect for custom models.
- **learning_rate** (float) – The learning rate to use for training

41.4.2 InstanceSegmentationData

class flash.image.instance_segmentation.data.**InstanceSegmentationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

classmethod from_coco(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *InstanceSegmentationData* object from the given data folders and annotation files in the COCO format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.

- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = InstanceSegmentationData.from_coco(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

classmethod from_voc(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *InstanceSegmentationData* object from the given data folders and annotation files in the VOC format.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.

- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Returns The constructed data module.

Examples:

```
data_module = InstanceSegmentationData.from_voc(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

41.4.3 InstanceSegmentationPreprocess

class flash.image.instance_segmentation.data.**InstanceSegmentationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(128, 128)*, *parser=None*)

41.5 Embedding

ImageEmbedder	The `ImageEmbedder` is a `Task` for obtaining feature vectors (embeddings) from images.

41.5.1 ImageEmbedder

class flash.image.embedding.model.**ImageEmbedder**(*embedding_dim=None*, *backbone='resnet101'*, *pretrained=True*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.SGD*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=torchmetrics.Accuracy*, *learning_rate=0.001*, *pooling_fn=torch.max*)

The `ImageEmbedder` is a `Task` for obtaining feature vectors (embeddings) from images. For more details, see *Image Embedder*.

Parameters

- **embedding_dim** (Optional[int]) – Dimension of the embedded vector. `None` uses the default from the backbone.
- **backbone** (str) – A model to use to extract image features, defaults to `"swav-imagenet"`.
- **pretrained** (bool) – Use a pretrained backbone, defaults to `True`.
- **loss_fn** (Callable) – Loss function for training and finetuning, defaults to `torch.nn.functional.cross_entropy()`

- **optimizer** (Type[Optimizer]) – Optimizer to use for training and finetuning, defaults to torch.optim.SGD.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inherenting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to torchmetrics.Accuracy.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.
- **pooling_fn** (Callable) – Function used to pool image to generate embeddings, defaults to torch.max().

41.6 Segmentation

SemanticSegmentation	SemanticSegmentation is a Task for semantic segmentation of images.
SemanticSegmentationData	Data module for semantic segmentation tasks.
SemanticSegmentationPreprocess	
segmentation.data.SegmentationMatplotlibVisualization	Process and show the image batch and its associated label using matplotlib.
segmentation.data.SemanticSegmentationNumpyDataSource	
segmentation.data.SemanticSegmentationTensorDataSource	
segmentation.data.SemanticSegmentationPathsDataSource	
segmentation.data.SemanticSegmentationFiftyOneDataSource	
segmentation.data.SemanticSegmentationDeserializer	
segmentation.model.SemanticSegmentationPostprocess	
segmentation.serialization.FiftyOneSegmentationLabels	A *Serializer* which converts the model outputs to FiftyOne segmentation format.
segmentation.serialization.SegmentationLabels	A *Serializer* which converts the model outputs to the label of the argmax classification per pixel in the image for semantic segmentation tasks.

41.6.1 SemanticSegmentation

class flash.image.segmentation.model.**SemanticSegmentation**(*num_classes*, *backbone='resnet50'*, *backbone_kwargs=None*, *head='fpn'*, *head_kwargs=None*, *pretrained=True*, *loss_fn=None*, *optimizer=torch.optim.AdamW*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.001*, *multi_label=False*, *serializer=None*, *postprocess=None*)

SemanticSegmentation is a Task for semantic segmentation of images. For more details, see *Semantic Segmentation*.

Parameters

- **num_classes** (int) – Number of classes to classify.
- **backbone** (Union[str, Module]) – A string or model to use to compute image features.
- **backbone_kwargs** (Optional[Dict]) – Additional arguments for the backbone configuration.
- **head** (str) – A string or (model, num_features) tuple to use to compute image features.
- **head_kwargs** (Optional[Dict]) – Additional arguments for the head configuration.
- **pretrained** (Union[bool, str]) – Use a pretrained backbone.
- **loss_fn** (Optional[Callable]) – Loss function for training.
- **optimizer** (Type[Optimizer]) – Optimizer to use for training.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inherenting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to torchmetrics.IOU.
- **learning_rate** (float) – Learning rate to use for training.
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

41.6.2 SemanticSegmentationData

class flash.image.segmentation.data.**SemanticSegmentationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for semantic segmentation tasks.

classmethod from_folders(*train_folder=None*, *train_target_folder=None*, *val_folder=None*, *val_target_folder=None*, *test_folder=None*, *test_target_folder=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *num_classes=None*, *labels_map=None*, ***preprocess_kwargs*)

Creates a *SemanticSegmentationData* object from the given data folders and corresponding target folders.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_target_folder** (Optional[str]) – The folder containing the train targets (targets must have the same file name as their corresponding inputs).
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_target_folder** (Optional[str]) – The folder containing the validation targets (targets must have the same file name as their corresponding inputs).
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_target_folder** (Optional[str]) – The folder containing the test targets (targets must have the same file name as their corresponding inputs).
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.

- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **num_classes** (Optional[int]) – Number of classes within the segmentation mask.
- **labels_map** (Optional[Dict[int, Tuple[int, int, int]]]) – Mapping between a class_id and its corresponding color.
- **preprocess_kwargs** – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = SemanticSegmentationData.from_folders(
    train_folder="train_folder",
    train_target_folder="train_masks",
)
```

set_block_viz_window(*value*)
Setter method to switch on/off matplotlib to pop up windows.

Return type None

41.6.3 SemanticSegmentationPreprocess

class flash.image.segmentation.data.**SemanticSegmentationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(128, 128)*, *deserializer=None*, *num_classes=None*, *labels_map=None*, ***data_source_kwargs*)

41.6.4 SegmentationMatplotlibVisualization

class flash.image.segmentation.data.**SegmentationMatplotlibVisualization**(*labels_map*)
Process and show the image batch and its associated label using matplotlib.

41.6.5 SemanticSegmentationNumpyDataSource

class flash.image.segmentation.data.**SemanticSegmentationNumpyDataSource**(*labels=None*)

41.6.6 SemanticSegmentationTensorDataSource

class flash.image.segmentation.data.**SemanticSegmentationTensorDataSource**(*labels=None*)

41.6.7 SemanticSegmentationPathsDataSource

class flash.image.segmentation.data.**SemanticSegmentationPathsDataSource**

41.6.8 SemanticSegmentationFiftyOneDataSource

class flash.image.segmentation.data.**SemanticSegmentationFiftyOneDataSource**(*label_field='ground_truth'*)

41.6.9 SemanticSegmentationDeserializer

class flash.image.segmentation.data.**SemanticSegmentationDeserializer**

41.6.10 SemanticSegmentationPostprocess

class flash.image.segmentation.model.**SemanticSegmentationPostprocess**(*save_path=None*)

41.6.11 FiftyOneSegmentationLabels

class flash.image.segmentation.serialization.**FiftyOneSegmentationLabels**(*labels_map=None*, *visualize=False*, *return_filepath=False*)

A *Serializer* which converts the model outputs to FiftyOne segmentation format.

Parameters

- **labels_map** (Optional[Dict[int, Tuple[int, int, int]]]) – A dictionary that map the labels ids to pixel intensities.
- **visualize** (bool) – whether to visualize the image labels.
- **return_filepath** (bool) – Boolean determining whether to return a dict containing filepath and FiftyOne labels (True) or only a list of FiftyOne labels (False).

41.6.12 SegmentationLabels

class flash.image.segmentation.serialization.**SegmentationLabels**(*labels_map=None*, *visualize=False*)

A *Serializer* which converts the model outputs to the label of the argmax classification per pixel in the image for semantic segmentation tasks.

Parameters

- **labels_map** (Optional[Dict[int, Tuple[int, int, int]]]) – A dictionary that map the labels ids to pixel intensities.
- **visualize** (bool) – Wether to visualize the image labels.

static **labels_to_image**(*img_labels*, *labels_map*)

Function that given an image with labels ids and their pixels intrensity mapping, creates a RGB representation for visualisation purposes.

Return type Tensor

segmentation.transforms.default_transforms	The default transforms for semantic segmentation: resize the image and mask, collate the batch, and apply normalization.
segmentation.transforms.prepare_target	Convert the target mask to long and remove the channel dimension.
segmentation.transforms.train_default_transforms	During training, we apply the default transforms with additional RandomHorizontalFlip and ColorJitter.

41.6.13 flash.image.segmentation.transforms.default_transforms

flash.image.segmentation.transforms.**default_transforms**(*image_size*)

The default transforms for semantic segmentation: resize the image and mask, collate the batch, and apply normalization.

Return type Dict[str, Callable]

41.6.14 flash.image.segmentation.transforms.prepare_target

flash.image.segmentation.transforms.**prepare_target**(*tensor*)

Convert the target mask to long and remove the channel dimension.

Return type Tensor

41.6.15 flash.image.segmentation.transforms.train_default_transforms

flash.image.segmentation.transforms.**train_default_transforms**(*image_size*)

During training, we apply the default transforms with additional RandomHorizontalFlip and ColorJitter.

Return type Dict[str, Callable]

41.7 Style Transfer

StyleTransfer	StyleTransfer is a Task for transferring the style from one image onto another.
StyleTransferData	
StyleTransferPreprocess	

41.7.1 StyleTransfer

class flash.image.style_transfer.model.**StyleTransfer**(*style_image=None*, *model=None*, *backbone='vgg16'*, *content_layer='relu2_2'*, *content_weight=100000.0*, *style_layers=['relu1_2', 'relu2_2', 'relu3_3', 'relu4_3']*, *style_weight=10000000000.0*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=0.001*, *serializer=None*)

StyleTransfer is a Task for transferring the style from one image onto another. For more details, see *Style Transfer*.

Parameters

- **style_image** (Union[str, Tensor, None]) – Image or path to an image to derive the style from.
- **model** (Optional[Module]) – The model by the style transfer task.
- **backbone** (str) – A string or model to use to compute the style loss from.
- **content_layer** (str) – Which layer from the backbone to extract the content loss from.
- **content_weight** (float) – The weight associated with the content loss. A lower value will lose content over style.
- **style_layers** (Union[Sequence[str], str]) – Layers from the backbone to derive the style loss from.
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training the model.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Optimizer keywords arguments.
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – Scheduler to use for training the model.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Scheduler keywords arguments.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

41.7.2 StyleTransferData

class flash.image.style_transfer.data.**StyleTransferData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

41.7.3 StyleTransferPreprocess

class flash.image.style_transfer.data.**StyleTransferPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=256*)

raise_not_supported

rtype NoReturn

41.7.4 flash.image.style_transfer.utils.raise_not_supported

flash.image.style_transfer.utils.**raise_not_supported**(*phase*)

Return type NoReturn

41.8 flash.image.data

ImageDeserializer

ImageFiftyOneDataSource

ImageNumpyDataSource

ImagePathsDataSource

ImageTensorDataSource

41.8.1 ImageDeserializer

class flash.image.data.**ImageDeserializer**

41.8.2 ImageFiftyOneDataSource

class flash.image.data.**ImageFiftyOneDataSource**(*label_field='ground_truth'*)

41.8.3 ImageNumpyDataSource

class flash.image.data.**ImageNumpyDataSource**(*labels=None*)

41.8.4 ImagePathsDataSource

class flash.image.data.**ImagePathsDataSource**

41.8.5 ImageTensorDataSource

class flash.image.data.**ImageTensorDataSource**(*labels=None*)

CHAPTER

FORTYTWO

FLASH.AUDIO

42.1 Classification

`AudioClassificationData`	Data module for audio classification.
`AudioClassificationPreprocess`	

42.1.1 AudioClassificationData

class flash.audio.classification.data.**AudioClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for audio classification.

42.1.2 AudioClassificationPreprocess

class flash.audio.classification.data.**AudioClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *spectrogram_size=(128, 128)*, *time_mask_param=None*, *freq_mask_param=None*, *deserializer=None*)

42.2 Speech Recognition

SpeechRecognitionData	Data Module for text classification tasks.
SpeechRecognition	The `SpeechRecognition` task is a `Task` for converting speech to text.
speech_recognition.data.SpeechRecognitionPreprocess	
speech_recognition.data.SpeechRecognitionBackboneState	The `SpeechRecognitionBackboneState` stores the backbone in use by the *SpeechRecognitionPostprocess*
speech_recognition.data.SpeechRecognitionPostprocess	
speech_recognition.data.SpeechRecognitionCSVDataSource	
speech_recognition.data.SpeechRecognitionJSONDataSource	
speech_recognition.data.BaseSpeechRecognition	
speech_recognition.data.SpeechRecognitionFileDataSource	
speech_recognition.data.SpeechRecognitionPathsDataSource	
speech_recognition.data.SpeechRecognitionDatasetDataSource	
speech_recognition.data.SpeechRecognitionDeserializer	

42.2.1 SpeechRecognitionData

class flash.audio.speech_recognition.data.**SpeechRecognitionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data Module for text classification tasks.

42.2.2 SpeechRecognition

class flash.audio.speech_recognition.model.**SpeechRecognition**(*backbone='facebook/wav2vec2-base-960h'*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=1e-05*, *serializer=None*)

The `SpeechRecognition` task is a `Task` for converting speech to text. For more details, see *Speech Recognition*.

Parameters

- **backbone** (`str`) – Any speech recognition model from HuggingFace/transformers.
- **optimizer** (`Type[Optimizer]`) – Optimizer to use for training.
- **optimizer_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (`Union[Type[LRScheduler], str, LRScheduler, None`]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **learning_rate** (`float`) – Learning rate to use for training, defaults to `1e-3`.
- **serializer** (`Union[Serializer, Mapping[str, Serializer], None]`) – The *Serializer* to use when serializing prediction outputs.

42.2.3 SpeechRecognitionPreprocess

class flash.audio.speech_recognition.data.**SpeechRecognitionPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *sampling_rate=16000*)

42.2.4 SpeechRecognitionBackboneState

class flash.audio.speech_recognition.data.**SpeechRecognitionBackboneState**(*backbone*)

The SpeechRecognitionBackboneState stores the backbone in use by the *SpeechRecognitionPostprocess*

42.2.5 SpeechRecognitionPostprocess

class flash.audio.speech_recognition.data.**SpeechRecognitionPostprocess**

42.2.6 SpeechRecognitionCSVDataSource

class flash.audio.speech_recognition.data.**SpeechRecognitionCSVDataSource**(*sampling_rate*)

42.2.7 SpeechRecognitionJSONDataSource

class flash.audio.speech_recognition.data.**SpeechRecognitionJSONDataSource**(*sampling_rate*)

42.2.8 BaseSpeechRecognition

class flash.audio.speech_recognition.data.**BaseSpeechRecognition**

42.2.9 SpeechRecognitionFileDataSource

class flash.audio.speech_recognition.data.**SpeechRecognitionFileDataSource**(*sampling_rate*, *filetype=None*)

42.2.10 SpeechRecognitionPathsDataSource

class flash.audio.speech_recognition.data.**SpeechRecognitionPathsDataSource**(*sampling_rate*)

42.2.11 SpeechRecognitionDatasetDataSource

class flash.audio.speech_recognition.data.**SpeechRecognitionDatasetDataSource**(*sampling_rate*)

42.2.12 SpeechRecognitionDeserializer

class flash.audio.speech_recognition.data.**SpeechRecognitionDeserializer**(*sampling_rate*)

CHAPTER

FORTYTHREE

FLASH.POINTCLOUD

43.1 Segmentation

PointCloudSegmentation	The PointCloudClassifier is a *ClassificationTask* that classifies pointcloud data.
PointCloudSegmentationData	
segmentation.data.PointCloudSegmentationPreprocess	
segmentation.data.PointCloudSegmentationFoldersDataSource	
segmentation.data.PointCloudSegmentationDatasetDataSource	

43.1.1 PointCloudSegmentation

class flash.pointcloud.segmentation.model.**PointCloudSegmentation**(*num_classes*, *backbone='RandLANet'*, *backbone_kwargs=None*, *head=None*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.01*, *multi_label=False*, *serializer=<flash.pointcloud.segmentation.model.PointC object>*)

The PointCloudClassifier is a *ClassificationTask* that classifies pointcloud data.

Parameters

- **num_features** – The number of features (elements) in the input data.
- **num_classes** (int) – The number of classes (outputs) for this *Task*.
- **backbone** (Union[str, Tuple[Module, int]]) – The backbone name (or a tuple of nn.Module, output size) to use.
- **backbone_kwargs** (Optional[Dict]) – Any additional kwargs to pass to the backbone constructor.
- **loss_fn** (Optional[Callable]) – The loss function to use. If None, a default will be selected by the *ClassificationTask* depending on the multi_label argument.
- **optimizer** (Union[Type[Optimizer], Optimizer]) – The optimizer or optimizer class to use.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Any metrics to use with this *Task*. If None, a default will be selected by the *ClassificationTask* depending on the multi_label argument.
- **learning_rate** (float) – The learning rate for the optimizer.
- **multi_label** (bool) – If True, this will be treated as a multi-label classification problem.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use for prediction outputs.

forward(*x*)

First call the backbone, then the model head.

Return type Tensor

43.1.2 PointCloudSegmentationData

class flash.pointcloud.segmentation.data.**PointCloudSegmentationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

43.1.3 PointCloudSegmentationPreprocess

class flash.pointcloud.segmentation.data.**PointCloudSegmentationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(196, 196)*, *deserializer=None*)

43.1.4 PointCloudSegmentationFoldersDataSource

class flash.pointcloud.segmentation.data.**PointCloudSegmentationFoldersDataSource**

43.1.5 PointCloudSegmentationDatasetDataSource

class flash.pointcloud.segmentation.data.**PointCloudSegmentationDatasetDataSource**

43.2 Object Detection

PointCloudObjectDetector	The PointCloudObjectDetector is a *ClassificationTask* that classifies pointcloud data.
PointCloudObjectDetectorData	
detection.data.PointCloudObjectDetectorPreprocess	
detection.data.PointCloudObjectDetectorFoldersDataSource	
detection.data.PointCloudObjectDetectorDatasetDataSource	

43.2.1 PointCloudObjectDetector

class flash.pointcloud.detection.model.**PointCloudObjectDetector**(*num_classes*, *backbone='pointpillars_kitti'*, *backbone_kwargs=None*, *head=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.01*, *serializer=<flash.pointcloud.detection.model.PointCloud object>*, *lambda_loss_cls=1.0*, *lambda_loss_bbox=1.0*, *lambda_loss_dir=1.0*)

The PointCloudObjectDetector is a *ClassificationTask* that classifies pointcloud data.

Parameters

- **num_features** – The number of features (elements) in the input data.
- **num_classes** (int) – The number of classes (outputs) for this *Task*.
- **backbone** (Union[str, Tuple[Module, int]]) – The backbone name (or a tuple of nn.Module, output size) to use.
- **backbone_kwargs** (Optional[Dict]) – Any additional kwargs to pass to the backbone constructor.
- **loss_fn** (Optional[Callable]) – The loss function to use. If None, a default will be selected by the *ClassificationTask* depending on the multi_label argument.
- **optimizer** (Union[Type[Optimizer], Optimizer]) – The optimizer or optimizer class to use.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Any metrics to use with this *Task*. If None, a default will be selected by the *ClassificationTask* depending on the multi_label argument.
- **learning_rate** (float) – The learning rate for the optimizer.
- **multi_label** – If True, this will be treated as a multi-label classification problem.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use for prediction outputs.
- **lambda_loss_cls** (float) – The value to scale the loss classification.
- **lambda_loss_bbox** (float) – The value to scale the bounding boxes loss.
- **lambda_loss_dir** (float) – The value to scale the bounding boxes direction loss.

forward(*x*)
First call the backbone, then the model head.

Return type Tensor

43.2.2 PointCloudObjectDetectorData

class flash.pointcloud.detection.data.**PointCloudObjectDetectorData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

classmethod from_folders(*train_folder=None*, *val_folder=None*, *test_folder=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, *scans_folder_name='scans'*, *labels_folder_name='labels'*, *calibrations_folder_name='calibs'*, *data_format='kitti'*, ***preprocess_kwargs*)
Creates a *DataModule* object from the given folders using the *DataSource* of name FOLDERS from the passed or constructed *Preprocess*.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.

- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **sampler** (Optional[Type[Sampler]]) – The sampler to use for the train_dataloader.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.
- **scans_folder_name** (Optional[str]) – The name of the pointcloud scan folder
- **labels_folder_name** (Optional[str]) – The name of the pointcloud scan labels folder
- **calibrations_folder_name** (Optional[str]) – The name of the pointcloud scan calibration folder
- **data_format** (Optional[BaseDataFormat]) – Format in which the data are stored.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_folders(
    train_folder="train_folder",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

43.2.3 PointCloudObjectDetectorPreprocess

class flash.pointcloud.detection.data.**PointCloudObjectDetectorPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *deserializer=None*, ***data_source_kwargs*)

43.2.4 PointCloudObjectDetectorFoldersDataSource

class flash.pointcloud.detection.data.**PointCloudObjectDetectorFoldersDataSource**(*data_format=None*, *image_size=(375, 1242)*, ***loader_kwargs*)

43.2.5 PointCloudObjectDetectorDatasetDataSource

class flash.pointcloud.detection.data.**PointCloudObjectDetectorDatasetDataSource**(***kwargs*)

CHAPTER

FORTYFOUR

FLASH.TABULAR

44.1 Classification

TabularClassifier	The `TabularClassifier` is a `Task` for classifying tabular data.
TabularClassificationData	

44.1.1 TabularClassifier

class flash.tabular.classification.model.**TabularClassifier**(*num_features*, *num_classes*, *embedding_sizes=None*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.01*, *multi_label=False*, *serializer=None*, ***tabnet_kwargs*)

The `TabularClassifier` is a `Task` for classifying tabular data. For more details, see *Tabular Classification*.

Parameters

- **num_features** (`int`) – Number of columns in table (not including target column).
- **num_classes** (`int`) – Number of classes to classify.
- **embedding_sizes** (`Optional[List[Tuple[int, int]]]`) – List of (num_classes, emb_dim) to form categorical embeddings.
- **loss_fn** (`Callable`) – Loss function for training, defaults to cross entropy.

- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inherenting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to `torchmetrics.Accuracy`.
- **learning_rate** (float) – Learning rate to use for training, defaults to *1e-3*
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.
- ****tabnet_kwargs** – Optional additional arguments for the TabNet model, see pytorch_tabnet.

44.1.2 TabularClassificationData

class flash.tabular.classification.data.**TabularClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

44.2 Regression

TabularRegressionData

44.2.1 TabularRegressionData

class flash.tabular.regression.data.**TabularRegressionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

44.3 flash.tabular.data

TabularData	Data module for tabular tasks.
TabularDataFrameDataSource	
TabularCSVDataSource	
TabularDeserializer	
TabularPreprocess	
TabularPostprocess	

44.3.1 TabularData

class flash.tabular.data.**TabularData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for tabular tasks.

property embedding_sizes: list

Recommended embedding sizes.

Return type list

classmethod from_csv(*categorical_fields*, *numerical_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *TabularData* object from the given CSV files.

Parameters

- **categorical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing categorical inputs.
- **numerical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing numerical inputs.
- **target_fields** (Optional[str]) – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (Optional[str]) – The CSV file containing the training data.
- **val_file** (Optional[str]) – The CSV file containing the validation data.
- **test_file** (Optional[str]) – The CSV file containing the testing data.
- **predict_file** (Optional[str]) – The CSV file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.

- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = TabularData.from_csv(
    "categorical_input",
    "numerical_input",
    "target",
    train_file="train_data.csv",
)
```

classmethod from_data_frame(*categorical_fields*, *numerical_fields*, *target_fields=None*, *train_data_frame=None*, *val_data_frame=None*, *test_data_frame=None*, *predict_data_frame=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *TabularData* object from the given data frames.

Parameters

- **categorical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing categorical inputs.
- **numerical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing numerical inputs.
- **target_fields** (Optional[str]) – The field or fields (columns) in the CSV file to use for the target.
- **train_data_frame** (Optional[object]) – The pandas `DataFrame` containing the training data.
- **val_data_frame** (Optional[object]) – The pandas `DataFrame` containing the validation data.

- **test_data_frame** (Optional[object]) – The pandas DataFrame containing the testing data.
- **predict_data_frame** (Optional[object]) – The pandas DataFrame containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Returns The constructed data module.

Examples:

```
data_module = TabularData.from_data_frame(
    "categorical_input",
    "numerical_input",
    "target",
    train_data_frame=train_data,
)
```

44.3.2 TabularDataFrameDataSource

class flash.tabular.data.**TabularDataFrameDataSource**(*cat_cols=None*, *num_cols=None*, *target_col=None*, *mean=None*, *std=None*, *codes=None*, *target_codes=None*, *classes=None*, *is_regression=True*)

44.3.3 TabularCSVDataSource

class flash.tabular.data.**TabularCSVDataSource**(*cat_cols=None*, *num_cols=None*, *target_col=None*, *mean=None*, *std=None*, *codes=None*, *target_codes=None*, *classes=None*, *is_regression=True*)

44.3.4 TabularDeserializer

class flash.tabular.data.**TabularDeserializer**(*cat_cols=None*, *num_cols=None*, *target_col=None*, *mean=None*, *std=None*, *codes=None*, *target_codes=None*, *classes=None*, *is_regression=True*)

44.3.5 TabularPreprocess

class flash.tabular.data.**TabularPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *cat_cols=None*, *num_cols=None*, *target_col=None*, *mean=None*, *std=None*, *codes=None*, *target_codes=None*, *classes=None*, *is_regression=True*, *deserializer=None*)

44.3.6 TabularPostprocess

class flash.tabular.data.**TabularPostprocess**(*save_path=None*)

CHAPTER

FORTYFIVE

FLASH.TEXT

45.1 Classification

TextClassifier	The `TextClassifier` is a `Task` for classifying text.
TextClassificationData	Data Module for text classification tasks.
classification.data.TextClassificationPostprocess	
classification.data.TextClassificationPreprocess	
classification.data.TextCSVDataSource	
classification.data.TextDataSource	
classification.data.TextDeserializer	
classification.data.TextFileDataSource	
classification.data.TextJSONDataSource	
classification.data.TextSentencesDataSource	

45.1.1 TextClassifier

class flash.text.classification.model.**TextClassifier**(*num_classes*, *backbone='prajjwal1/bert-medium'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.01*, *multi_label=False*, *serializer=None*, *enable_ort=False*)

The `TextClassifier` is a `Task` for classifying text. For more details, see *Text Classification*. The `TextClassifier` also supports multi-label classification with `multi_label=True`. For more details, see *Multi-label Text Classification*.

Parameters

- **num_classes** (`int`) – Number of classes to classify.
- **backbone** (`str`) – A model to use to compute text features can be any BERT model from HuggingFace/transformersimage .
- **optimizer** (`Type[Optimizer]`) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **optimizer_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (`Union[Type[LRScheduler], str, LRScheduler, None`) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (`Union[Metric, Callable, Mapping, Sequence, None`) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inherenting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to `torchmetrics.Accuracy`.
- **learning_rate** (`float`) – Learning rate to use for training, defaults to *1e-3*
- **multi_label** (`bool`) – Whether the targets are multi-label or not.
- **serializer** (`Union[Serializer, Mapping[str, Serializer], None`) – The *Serializer* to use when serializing prediction outputs.
- **enable_ort** (`bool`) – Enable Torch ONNX Runtime Optimization: https://onnxruntime.ai/docs/#onnx-runtime-for-training

45.1.2 TextClassificationData

class flash.text.classification.data.**TextClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data Module for text classification tasks.

45.1.3 TextClassificationPostprocess

class flash.text.classification.data.**TextClassificationPostprocess**(*save_path=None*)

45.1.4 TextClassificationPreprocess

class flash.text.classification.data.**TextClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *backbone='prajjwal1/bert-tiny'*, *max_length=128*)

collate(*samples*)

Override to convert a set of samples to a batch.

Return type Tensor

45.1.5 TextCSVDataSource

class flash.text.classification.data.**TextCSVDataSource**(*backbone*, *max_length=128*)

45.1.6 TextDataSource

class flash.text.classification.data.**TextDataSource**(*backbone*, *max_length=128*)

45.1.7 TextDeserializer

class flash.text.classification.data.**TextDeserializer**(*backbone*, *max_length*, *use_fast=True*, ***kwargs*)

45.1.8 TextFileDataSource

class flash.text.classification.data.**TextFileDataSource**(*filetype*, *backbone*, *max_length=128*)

45.1.9 TextJSONDataSource

class flash.text.classification.data.**TextJSONDataSource**(*backbone*, *max_length=128*)

45.1.10 TextSentencesDataSource

class flash.text.classification.data.**TextSentencesDataSource**(*backbone*, *max_length=128*)

45.2 Question Answering

QuestionAnsweringTask	The `QuestionAnsweringTask` is a `Task` for extractive question answering.
QuestionAnsweringData	Data module for QuestionAnswering task.
question_answering.data.QuestionAnsweringBackboneState	The `QuestionAnsweringBackboneState` stores the backbone in use by the *QuestionAnsweringPreprocess*
question_answering.data.QuestionAnsweringCSVDataSource	
question_answering.data.QuestionAnsweringDataSource	
question_answering.data.QuestionAnsweringDictionaryDataSource	
question_answering.data.QuestionAnsweringFileDataSource	
question_answering.data.QuestionAnsweringJSONDataSource	
question_answering.data.QuestionAnsweringPostprocess	
question_answering.data.QuestionAnsweringPreprocess	
question_answering.data.SQuADDataSource	

45.2.1 QuestionAnsweringTask

class flash.text.question_answering.model.**QuestionAnsweringTask**(*backbone='distilbert-base-uncased'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=5e-05*, *enable_ort=False*, *n_best_size=20*, *version_2_with_negative=True*, *max_answer_length=30*, *null_score_diff_threshold=0.0*, *use_stemmer=True*, *rouge_newline_sep=True*)

The `QuestionAnsweringTask` is a `Task` for extractive question answering. For more details, see *question_answering*.

You can change the backbone to any question answering model from HuggingFace/transformers using the `backbone` argument.

Note: When changing the backbone, make sure you pass in the same backbone to the `Task` and the *DataModule* object! Since this is a QuestionAnswering task, make sure you use a QuestionAnswering model.

Parameters

- **backbone** (`str`) – backbone model to use for the task.
- **loss_fn** (`Union[Callable, Mapping, Sequence, None]`) – Loss function for training.
- **optimizer** (`Type[Optimizer]`) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **optimizer_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (`Union[Type[LRScheduler], str, LRScheduler, None]`) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (`Union[Metric, Callable, Mapping, Sequence, None]`) – Metrics to compute for training and evaluation. Defauls to calculating the ROUGE metric. Changing this argument currently has no effect.
- **learning_rate** (`float`) – Learning rate to use for training, defaults to *3e-4*
- **enable_ort** (`bool`) – Enable Torch ONNX Runtime Optimization: https://onnxruntime.ai/docs/#onnx-runtime-for-training
- **n_best_size** (`int`) – The total number of n-best predictions to generate when looking for an answer.
- **version_2_with_negative** (`bool`) – If true, some of the examples do not have an answer.

- **max_answer_length** (int) – The maximum length of an answer that can be generated. This is needed because the start and end predictions are not conditioned on one another.
- **null_score_diff_threshold** (float) – The threshold used to select the null answer: if the best answer has a score that is less than the score of the null answer minus this threshold, the null answer is selected for this example. Only useful when *version_2_with_negative=True*.
- **use_stemmer** (bool) – Whether Porter stemmer should be used to strip word suffixes to improve matching.
- **rouge_newline_sep** (bool) – Add a new line at the beginning of each sentence in Rouge Metric calculation.

property task: Optional[str]

Override to define AutoConfig task specific parameters stored within the model.

Return type Optional[str]

45.2.2 QuestionAnsweringData

class flash.text.question_answering.data.**QuestionAnsweringData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for QuestionAnswering task.

classmethod from_csv(*train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given CSV files using the *DataSource* of name CSV from the passed or constructed *Preprocess*.

Parameters

- **input_fields** – The field or fields (columns) in the CSV file to use for the input.
- **target_fields** – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (Optional[str]) – The CSV file containing the training data.
- **val_file** (Optional[str]) – The CSV file containing the validation data.
- **test_file** (Optional[str]) – The CSV file containing the testing data.
- **predict_file** (Optional[str]) – The CSV file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.

- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (Optional[Sampler]) – The `sampler` argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Note: The following keyword arguments can be passed through to the preprocess_kwargs

- backbone: The HF model to be used for the task.
- max_source_length: Max length of the sequence to be considered during tokenization.
- max_target_length: Max length of each answer to be produced.
- padding: Padding type during tokenization. Defaults to 'max_length'.
- question_column_name: The key in the JSON file to recognize the question field. Defaults to"question".
- context_column_name: The key in the JSON file to recognize the context field. Defaults to "context".
- answer_column_name: The key in the JSON file to recognize the answer field. Defaults to "answer".
- doc_stride: The stride amount to be taken when splitting up a long document into chunks.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = QuestionAnsweringData.from_csv(
    "input",
    "target",
    train_file="train_data.csv",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
    backbone="distilbert-base-uncased",
```

(continues on next page)

(continued from previous page)

```
    max_source_length=384,
    max_target_length=30,
    padding='max_length',
    question_column_name="question",
    context_column_name="context",
    answer_column_name="answer",
    doc_stride=128
)
```

classmethod from_json(*train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*, *field=None*, ***preprocess_kwargs*)

Creates a `QuestionAnsweringData` object from the given JSON files using the `JSON` from the passed or constructed `QuestionAnsweringPreprocess`.

Parameters

- **train_file** (`Optional[str]`) – The JSON file containing the training data.
- **val_file** (`Optional[str]`) – The JSON file containing the validation data.
- **test_file** (`Optional[str]`) – The JSON file containing the testing data.
- **predict_file** (`Optional[str]`) – The JSON file containing the data to use when predicting.
- **train_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (`Optional[BaseDataFetcher]`) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (`Optional[Preprocess]`) – The `Preprocess` to pass to the *DataModule*. If `None`, `cls.preprocess_cls` will be constructed and used.
- **val_split** (`Optional[float]`) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (`int`) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (`Optional[int]`) – The `num_workers` argument to pass to the *DataModule*.
- **sampler** (`Optional[Sampler]`) – The `sampler` argument to pass to the *DataModule*.
- **field** (`Optional[str]`) – To specify the field that holds the data in the JSON file.
- **preprocess_kwargs** (`Any`) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Note: The following keyword arguments can be passed through to the preprocess_kwargs

- backbone: The HF model to be used for the task.
- max_source_length: Max length of the sequence to be considered during tokenization.
- max_target_length: Max length of each answer to be produced.
- padding: Padding type during tokenization. Defaults to 'max_length'.
- question_column_name: The key in the JSON file to recognize the question field. Defaults to"question".
- context_column_name: The key in the JSON file to recognize the context field. Defaults to "context".
- answer_column_name: The key in the JSON file to recognize the answer field. Defaults to "answer".
- doc_stride: The stride amount to be taken when splitting up a long document into chunks.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = QuestionAnsweringData.from_json(
    train_file="train_data.json",
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
    backbone="distilbert-base-uncased",
    max_source_length=384,
    max_target_length=30,
    padding='max_length',
    question_column_name="question",
    context_column_name="context",
    answer_column_name="answer",
    doc_stride=128
)
```

classmethod from_squad_v2(*train_file=None*, *val_file=None*, *test_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *QuestionAnsweringData* object from the given data JSON files in the SQuAD2.0 format.

Parameters

- **train_file** (Optional[str]) – The JSON file containing the training data.
- **val_file** (Optional[str]) – The JSON file containing the validation data.
- **test_file** (Optional[str]) – The JSON file containing the testing data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.

- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, `cls.preprocess_cls` will be constructed and used.
- **val_split** (Optional[float]) – The `val_split` argument to pass to the *DataModule*.
- **batch_size** (int) – The `batch_size` argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The `num_workers` argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if `preprocess = None`.

Returns The constructed data module.

Examples:

```
data_module = QuestionAnsweringData.from_squad_v2(
    train_file="train.json",
    doc_stride=128,
)
```

45.2.3 QuestionAnsweringBackboneState

class flash.text.question_answering.data.**QuestionAnsweringBackboneState**(*backbone*)

The QuestionAnsweringBackboneState stores the backbone in use by the *QuestionAnsweringPreprocess*

45.2.4 QuestionAnsweringCSVDataSource

class flash.text.question_answering.data.**QuestionAnsweringCSVDataSource**(*backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.2.5 QuestionAnsweringDataSource

class flash.text.question_answering.data.**QuestionAnsweringDataSource**(*backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.2.6 QuestionAnsweringDictionaryDataSource

class flash.text.question_answering.data.**QuestionAnsweringDictionaryDataSource**(*backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.2.7 QuestionAnsweringFileDataSource

class flash.text.question_answering.data.**QuestionAnsweringFileDataSource**(*filetype*, *backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.2.8 QuestionAnsweringJSONDataSource

class flash.text.question_answering.data.**QuestionAnsweringJSONDataSource**(*backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.2.9 QuestionAnsweringPostprocess

class flash.text.question_answering.data.**QuestionAnsweringPostprocess**

static per_sample_transform(*sample*)

Transforms to apply to a single sample after splitting up the batch.

Can involve both CPU and Device transforms as this is not applied in separate workers.

Return type Any

45.2.10 QuestionAnsweringPreprocess

class flash.text.question_answering.data.**QuestionAnsweringPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *backbone='distilbert-base-uncased'*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

collate(*samples*)

Override to convert a set of samples to a batch.

Return type Tensor

45.2.11 SQuADDataSource

class flash.text.question_answering.data.**SQuADDataSource**(*backbone*, *max_source_length=384*, *max_target_length=30*, *padding='max_length'*, *question_column_name='question'*, *context_column_name='context'*, *answer_column_name='answer'*, *doc_stride=128*)

45.3 Summarization

SummarizationTask	The SummarizationTask is a Task for Seq2Seq text summarization.
SummarizationData	
seq2seq.summarization.data.SummarizationPreprocess	

45.3.1 SummarizationTask

class flash.text.seq2seq.summarization.model.**SummarizationTask**(*backbone='sshleifer/distilbart-xsum-1-1'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=1e-05*, *val_target_max_length=None*, *num_beams=4*, *use_stemmer=True*, *rouge_newline_sep=True*, *enable_ort=False*)

The SummarizationTask is a Task for Seq2Seq text summarization. For more details, see *Summarization*.

You can change the backbone to any summarization model from HuggingFace/transformers using the backbone argument.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the *DataModule* object! Since this is a Seq2Seq task, make sure you use a Seq2Seq model.

Parameters

- **backbone** (str) – backbone model to use for the task.
- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training.
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to *torch.optim.Adam*.

- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Defauls to calculating the ROUGE metric. Changing this argument currently has no effect.
- **learning_rate** (float) – Learning rate to use for training, defaults to *3e-4*
- **val_target_max_length** (Optional[int]) – Maximum length of targets in validation. Defaults to *128*
- **num_beams** (Optional[int]) – Number of beams to use in validation when generating predictions. Defaults to *4*
- **use_stemmer** (bool) – Whether Porter stemmer should be used to strip word suffixes to improve matching.
- **rouge_newline_sep** (bool) – Add a new line at the beginning of each sentence in Rouge Metric calculation.
- **enable_ort** (bool) – Enable Torch ONNX Runtime Optimization: https://onnxruntime.ai/docs/#onnx-runtime-for-training

45.3.2 SummarizationData

class flash.text.seq2seq.summarization.data.**SummarizationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

45.3.3 SummarizationPreprocess

class flash.text.seq2seq.summarization.data.**SummarizationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *backbone='sshleifer/distilbart-xsum-1-1'*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***kwargs*)

45.4 Translation

TranslationTask	The TranslationTask is a Task for Seq2Seq text translation.
TranslationData	Data module for Translation tasks.
seq2seq.translation.data.TranslationPreprocess	

45.4.1 TranslationTask

class flash.text.seq2seq.translation.model.**TranslationTask**(*backbone='t5-small'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=1e-05*, *val_target_max_length=128*, *num_beams=4*, *n_gram=4*, *smooth=True*, *enable_ort=False*)

The TranslationTask is a Task for Seq2Seq text translation. For more details, see *Translation*.

You can change the backbone to any translation model from HuggingFace/transformers using the backbone argument.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the *DataModule* object! Since this is a Seq2Seq task, make sure you use a Seq2Seq model.

Parameters

- **backbone** (str) – backbone model to use for the task.
- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training.
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Defauls to calculating the BLEU metric. Changing this argument currently has no effect.
- **learning_rate** (float) – Learning rate to use for training, defaults to *1e-5*
- **val_target_max_length** (Optional[int]) – Maximum length of targets in validation. Defaults to *128*

- **num_beams** (Optional[int]) – Number of beams to use in validation when generating predictions. Defaults to *4*
- **n_gram** (bool) – Maximum n_grams to use in metric calculation. Defaults to *4*
- **smooth** (bool) – Apply smoothing in BLEU calculation. Defaults to *True*
- **enable_ort** (bool) – Enable Torch ONNX Runtime Optimization: https://onnxruntime.ai/docs/#onnx-runtime-for-training

45.4.2 TranslationData

class flash.text.seq2seq.translation.data.**TranslationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for Translation tasks.

45.4.3 TranslationPreprocess

class flash.text.seq2seq.translation.data.**TranslationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *backbone='t5-small'*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***kwargs*)

45.5 General Seq2Seq

Seq2SeqTask	General Task for Sequence2Sequence.
Seq2SeqData	Data module for Seq2Seq tasks.
Seq2SeqFreezeEmbeddings	Freezes the embedding layers during Seq2Seq training.
seq2seq.core.data.Seq2SeqBackboneState	The `Seq2SeqBackboneState` stores the backbone in use by the *Seq2SeqPreprocess*
seq2seq.core.data.Seq2SeqCSVDataSource	
seq2seq.core.data.Seq2SeqDataSource	
seq2seq.core.data.Seq2SeqFileDataSource	
seq2seq.core.data.Seq2SeqJSONDataSource	
seq2seq.core.data.Seq2SeqPostprocess	

continues on next page

Table 5 – continued from previous page

seq2seq.core.data.Seq2SeqPreprocess	
seq2seq.core.data.Seq2SeqSentencesDataSource	
seq2seq.core.metrics.BLEUScore	Calculate BLEU score of machine translated text with one or more references.
seq2seq.core.metrics.RougeBatchAggregator	Aggregates rouge scores and provides confidence intervals.
seq2seq.core.metrics.RougeMetric	Metric used for automatic summarization.

45.5.1 Seq2SeqTask

class flash.text.seq2seq.core.model.**Seq2SeqTask**(*backbone='t5-small'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=5e-05*, *val_target_max_length=None*, *num_beams=None*, *enable_ort=False*)

General Task for Sequence2Sequence.

Parameters

- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Changing this argument currently has no effect
- **learning_rate** (float) – Learning rate to use for training, defaults to *3e-4*
- **val_target_max_length** (Optional[int]) – Maximum length of targets in validation. Defaults to *128*
- **num_beams** (Optional[int]) – Number of beams to use in validation when generating predictions. Defaults to *4*
- **enable_ort** (bool) – Enable Torch ONNX Runtime Optimization: https://onnxruntime.ai/docs/#onnx-runtime-for-training

property task: Optional[str]

Override to define AutoConfig task specific parameters stored within the model.

Return type Optional[str]

45.5.2 Seq2SeqData

class flash.text.seq2seq.core.data.**Seq2SeqData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for Seq2Seq tasks.

45.5.3 Seq2SeqFreezeEmbeddings

class flash.text.seq2seq.core.finetuning.**Seq2SeqFreezeEmbeddings**(*model_type*, *train_bn=True*)

Freezes the embedding layers during Seq2Seq training.

45.5.4 Seq2SeqBackboneState

class flash.text.seq2seq.core.data.**Seq2SeqBackboneState**(*backbone*, *backbone_kwargs=<factory>*)

The `Seq2SeqBackboneState` stores the backbone in use by the *Seq2SeqPreprocess*

45.5.5 Seq2SeqCSVDataSource

class flash.text.seq2seq.core.data.**Seq2SeqCSVDataSource**(*backbone*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

45.5.6 Seq2SeqDataSource

class flash.text.seq2seq.core.data.**Seq2SeqDataSource**(*backbone*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

45.5.7 Seq2SeqFileDataSource

class flash.text.seq2seq.core.data.**Seq2SeqFileDataSource**(*filetype*, *backbone*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

45.5.8 Seq2SeqJSONDataSource

class flash.text.seq2seq.core.data.**Seq2SeqJSONDataSource**(*backbone*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

45.5.9 Seq2SeqPostprocess

class flash.text.seq2seq.core.data.**Seq2SeqPostprocess**

45.5.10 Seq2SeqPreprocess

class flash.text.seq2seq.core.data.**Seq2SeqPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *backbone='sshleifer/tiny-mbart'*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

collate(*samples*)

Override to convert a set of samples to a batch.

Return type Tensor

45.5.11 Seq2SeqSentencesDataSource

class flash.text.seq2seq.core.data.**Seq2SeqSentencesDataSource**(*backbone*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, ***backbone_kwargs*)

45.5.12 BLEUScore

class flash.text.seq2seq.core.metrics.**BLEUScore**(*n_gram=4*, *smooth=False*)

Calculate BLEU score of machine translated text with one or more references.

Example

```
>>> translate_corpus = ['the cat is on the mat'.split()]
>>> reference_corpus = [['there is a cat on the mat'.split(), 'a cat is on the mat'.
↪split()]]
>>> metric = BLEUScore()
>>> metric(translate_corpus, reference_corpus)
tensor(0.7598)
```

update(*translate_corpus*, *reference_corpus*)
Actual metric computation :param _sphinx_paramlinks_flash.text.seq2seq.core.metrics.BLEUScore.update.translate_corpus:
An iterable of machine translated corpus :param _sphinx_paramlinks_flash.text.seq2seq.core.metrics.BLEUScore.update.refe
An iterable of iterables of reference corpus

Return type None

45.5.13 RougeBatchAggregator

class flash.text.seq2seq.core.metrics.**RougeBatchAggregator**
Aggregates rouge scores and provides confidence intervals.

aggregate()
Override function to wrap the final results in *Score* objects.

This is due to the scores being replaced with a list of torch tensors.

45.5.14 RougeMetric

class flash.text.seq2seq.core.metrics.**RougeMetric**(*rouge_newline_sep=False*, *use_stemmer=False*, *rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum')*)
Metric used for automatic summarization. https://www.aclweb.org/anthology/W04-1013/

Example

```
>>> target = "Is your name John".split()
>>> preds = "My name is John".split()
>>> rouge = RougeMetric()
>>> from pprint import pprint
>>> pprint(rouge(preds, target))
{'rouge1_fmeasure': 0.25,
 'rouge1_precision': 0.25,
 'rouge1_recall': 0.25,
 'rouge2_fmeasure': 0.0,
 'rouge2_precision': 0.0,
 'rouge2_recall': 0.0,
 'rougeL_fmeasure': 0.25,
 'rougeL_precision': 0.25,
 'rougeL_recall': 0.25,
 'rougeLsum_fmeasure': 0.25,
 'rougeLsum_precision': 0.25,
 'rougeLsum_recall': 0.25}
```

CHAPTER

FORTYSIX

FLASH.VIDEO

- *Classification*

46.1 Classification

VideoClassifier	Task that classifies videos.
VideoClassificationData	Data module for Video classification tasks.
classification.data.BaseVideoClassification	
classification.data.VideoClassificationFiftyOneDataSource	
classification.data.VideoClassificationPathsDataSource	
classification.data.VideoClassificationPreprocess	
classification.model.VideoClassifierFinetuning	

46.1.1 VideoClassifier

class flash.video.classification.model.**VideoClassifier**(*num_classes*, *backbone='x3d_xs'*, *backbone_kwargs=None*, *pretrained=True*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.SGD*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=torchmetrics.Accuracy*, *learning_rate=0.001*, *head=None*, *serializer=None*)

Task that classifies videos.

Parameters

- **num_classes** (int) – Number of classes to classify.
- **backbone** (Union[str, Module]) – A string mapped to pytorch_video backbones or nn.Module, defaults to "slowfast_r50".

- **backbone_kwargs** (Optional[Dict]) – Arguments to customize the backbone from PyTorchVideo.
- **pretrained** (bool) – Use a pretrained backbone, defaults to True.
- **loss_fn** (Callable) – Loss function for training, defaults to torch.nn.functional.cross_entropy().
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to torch.optim.SGD.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Metric, Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation. Can either be an metric from the *torchmetrics* package, a custom metric inherenting from *torchmetrics.Metric*, a callable function or a list/dict containing a combination of the aforementioned. In all cases, each metric needs to have the signature *metric(preds,target)* and return a single scalar tensor. Defaults to torchmetrics.Accuracy.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.

46.1.2 VideoClassificationData

class flash.video.classification.data.**VideoClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for Video classification tasks.

46.1.3 BaseVideoClassification

class flash.video.classification.data.**BaseVideoClassification**(*clip_sampler*, *video_sampler=torch.utils.data.RandomSampler*, *decode_audio=True*, *decoder='pyav'*)

46.1.4 VideoClassificationFiftyOneDataSource

class flash.video.classification.data.**VideoClassificationFiftyOneDataSource**(*clip_sampler*, *video_sampler=torch.utils.data.Ra* *de-* *code_audio=True*, *decoder='pyav'*, *la-* *bel_field='ground_truth'*)

46.1.5 VideoClassificationPathsDataSource

class flash.video.classification.data.**VideoClassificationPathsDataSource**(*clip_sampler*, *video_sampler=torch.utils.data.Random* *decode_audio=True*, *decoder='pyav'*)

46.1.6 VideoClassificationPreprocess

class flash.video.classification.data.**VideoClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *clip_sampler='random'*, *clip_duration=2*, *clip_sampler_kwargs=None*, *video_sampler=torch.utils.data.RandomSamp* *decode_audio=True*, *decoder='pyav'*, ***data_source_kwargs*)

46.1.7 VideoClassifierFinetuning

class flash.video.classification.model.**VideoClassifierFinetuning**(*num_layers=5*, *train_bn=True*, *unfreeze_epoch=1*)

CHAPTER

FORTYSEVEN

FLASH.GRAPH

47.1 Classification

GraphClassifier	The `GraphClassifier` is a `Task` for classifying graphs.
GraphClassificationData	Data module for graph classification tasks.
classification.data.GraphClassificationPreprocess	

47.1.1 GraphClassifier

class flash.graph.classification.model.**GraphClassifier**(*num_features*, *num_classes*, *hidden_channels=512*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.001*, *model=None*, *conv_cls=None*, ***conv_kwargs*)

The `GraphClassifier` is a `Task` for classifying graphs. For more details, see *Graph Classification*.

Parameters

- **num_features** (`int`) – Number of columns in table (not including target column).
- **num_classes** (`int`) – Number of classes to classify.
- **hidden_channels** (`Union[List[int], int]`) – Hidden dimension sizes.
- **loss_fn** (`Callable`) – Loss function for training, defaults to cross entropy.
- **optimizer** (`Type[Optimizer]`) – Optimizer to use for training, defaults to *torch.optim.Adam*.
- **optimizer_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the optimizer (if not passed as an instance).

- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (Union[Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **learning_rate** (float) – Learning rate to use for training, defaults to *1e-3*
- **model** (Optional[Module]) – GraphNN used, defaults to BaseGraphModel.
- **conv_cls** (Optional[Type[None]]) – kind of convolution used in model, defaults to GCNConv

47.1.2 GraphClassificationData

class flash.graph.classification.data.**GraphClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *sampler=None*)

Data module for graph classification tasks.

47.1.3 GraphClassificationPreprocess

class flash.graph.classification.data.**GraphClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*)

47.2 flash.graph.data

GraphDatasetDataSource

47.2.1 GraphDatasetDataSource

class flash.graph.data.**GraphDatasetDataSource**

CHAPTER

FORTYEIGHT

INTRODUCTION / SET-UP

48.1 Welcome

Before you begin, we'd like to express our gratitude to you for wanting to add a task to Flash. With Flash our aim is to create a great user experience, enabling awesome advanced applications with just a few lines of code. We're really pleased with what we've achieved with Flash and we hope you will be too. Now let's dive in!

48.2 Set-up

The Task template is designed to guide you through contributing a task to Flash. It contains the code, tests, and examples for a task that performs classification with a multi-layer perceptron, intended for use with the classic data sets from scikit-learn. The Flash tasks are organized in folders by data-type (image, text, video, etc.), with sub-folders for different task types (classification, regression, etc.).

Copy the files in flash/template/classification to a new sub-directory under the relevant data-type. If a data-type folder already exists for your task, then a task type sub-folder should be added containing the template files. If a data-type folder doesn't exist, then you will need to add that too. You should also copy the files from tests/template/classification to the corresponding data-type, task type folder in `tests`. For example, if you were adding an image classification task, you would do:

```
mkdir flash/image/classification
cp flash/template/classification/* flash/image/classification/
mkdir tests/image/classification
cp tests/template/classification/* tests/image/classification/
```

48.3 Tutorials

The tutorials in this section will walk you through all of the components you need to implement (or adapt from the template) for your custom task.

- *The Data*: our first tutorial goes over the best practices for implementing everything you need to connect data to your task
- *The Backbones*: the second tutorial shows you how to create an extensible backbone registry for your task
- *The Task*: now that we have the data and the models, in this tutorial we create our custom task
- *Optional Extras*: this tutorial covers some optional extras you can add if needed for your particular task
- *The Example*: this tutorial guides you through creating some simple examples showing your task in action

- *The Tests*: in this tutorial, we cover best practices for writing some tests for your new task
- *The Docs*: in our final tutorial, we provide a template for you to create the docs page for your task

CHAPTER

FORTYNINE

THE DATA

The first step to contributing a task is to implement the classes we need to load some data. Inside data.py you should implement:

1. some *DataSource* classes *(optional)*
2. a *Preprocess*
3. a *DataModule*
4. a *BaseVisualization* *(optional)*
5. a *Postprocess* *(optional)*

49.1 DataSource

The *DataSource* class contains the logic for data loading from different sources such as folders, files, tensors, etc. Every Flash *DataModule* can be instantiated with *from_datasets()*. For each additional way you want the user to be able to instantiate your *DataModule*, you'll need to create a *DataSource*. Each *DataSource* has 2 methods:

- *load_data()* takes some dataset metadata (e.g. a folder name) as input and produces a sequence or iterable of samples or sample metadata.
- *load_sample()* then takes as input a single element from the output of `load_data` and returns a sample.

By default these methods just return their input, so you don't need both a *load_data()* and a *load_sample()* to create a *DataSource*. Where possible, you should override one of our existing *DataSource* classes.

Let's start by implementing a `TemplateNumpyDataSource`, which overrides *NumpyDataSource*. The main *DataSource* method that we have to implement is *load_data()*. As we're extending the `NumpyDataSource`, we expect the same `data` argument (in this case, a tuple containing data and corresponding target arrays).

We can also take the dataset argument. Any attributes we set on `dataset` will be available on the `Dataset` generated by our *DataSource*. In this data source, we'll set the `num_features` attribute.

Here's the code for our `TemplateNumpyDataSource.load_data` method:

```
def load_data(self, data: Tuple[np.ndarray, Sequence[Any]], dataset: Any) ->␣
↪Sequence[Mapping[str, Any]]:
    """Sets the ``num_features`` attribute and calls ``super().load_data``.

    Args:
        data: The tuple of ``np.ndarray`` (num_examples x num_features) and associated␣
↪targets.
        dataset: The object that we can set attributes (such as ``num_features``) on.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        A sequence of samples / sample metadata.
    """
    dataset.num_features = data[0].shape[1]
    return super().load_data(data, dataset)
```

Note: Later, when we add *our DataModule implementation*, we'll make `num_features` available to the user.

Sometimes you need to something a bit more custom. When creating a custom *DataSource*, the type of the `data` argument is up to you. For our template `Task`, it would be cool if the user could provide a scikit-learn `Bunch` as the data source. To achieve this, we'll add a `TemplateSKLearnDataSource` whose `load_data` expects a `Bunch` as input. We override our `TemplateNumpyDataSource` so that we can call `super` with the data and targets extracted from the `Bunch`. We perform two additional steps here to improve the user experience:

1. We set the `num_classes` attribute on the `dataset`. If `num_classes` is set, it is automatically made available as a property of the *DataModule*.
2. We create and set a *LabelsState*. The labels provided here will be shared with the *Labels* serializer, so the user doesn't need to provide them.

Here's the code for the `TemplateSKLearnDataSource.load_data` method:

```
def load_data(self, data: Bunch, dataset: Any) -> Sequence[Mapping[str, Any]]:
    """Gets the ``data`` and ``target`` attributes from the ``Bunch`` and passes them to␣
↪``super().load_data``.

    Args:
        data: The scikit-learn data ``Bunch``.
        dataset: The object that we can set attributes (such as ``num_classes``) on.

    Returns:
        A sequence of samples / sample metadata.
    """
    dataset.num_classes = len(data.target_names)
    self.set_state(LabelsState(data.target_names))
    return super().load_data((data.data, data.target), dataset=dataset)
```

We can customize the behaviour of our *load_data()* for different stages, by prepending *train*, *val*, *test*, or *predict*. For our `TemplateSKLearnDataSource`, we don't want to provide any targets to the model when predicting. We can implement `predict_load_data` like this:

```
def predict_load_data(self, data: Bunch) -> Sequence[Mapping[str, Any]]:
    """Avoid including targets when predicting.

    Args:
        data: The scikit-learn data ``Bunch``.

    Returns:
        A sequence of samples / sample metadata.
    """
    return super().predict_load_data(data.data)
```

49.1.1 DataSource vs Dataset

A *DataSource* is not the same as a torch.utils.data.Dataset. When a from_* method is called on your *DataModule*, it gets the *DataSource* to use from the *Preprocess*. A Dataset is then created from the *DataSource* for each stage (*train*, *val*, *test*, *predict*) using the provided metadata (e.g. folder name, numpy array etc.).

The output of the *load_data()* can just be a torch.utils.data.Dataset instance. If the library that your Task is based on provides a custom dataset, you don't need to re-write it as a *DataSource*. For example, the *load_data()* of the VideoClassificationPathsDataSource just creates an EncodedVideoDataset from the given folder. Here's how it looks (from video/classification.data.py):

```
def load_data(self, data: str, dataset: Optional[Any] = None) -> "LabeledVideoDataset":
    ds = self._make_encoded_video_dataset(data)
    if self.training:
        label_to_class_mapping = {p[1]: p[0].split("/")[-2] for p in ds._labeled_videos._
↪paths_and_labels}
        self.set_state(LabelsState(label_to_class_mapping))
        dataset.num_classes = len(np.unique([s[1]["label"] for s in ds._labeled_videos]))
    return ds
```

49.2 Preprocess

The *Preprocess* object contains all the data transforms. Internally we inject the *Preprocess* transforms at several points along the pipeline.

Defining the standard transforms (typically at least a to_tensor_transform should be defined) for your *Preprocess* is as simple as implementing the default_transforms method. The *Preprocess* must take train_transform, val_transform, test_transform, and predict_transform arguments in the __init__. These arguments can be provided by the user (when creating the *DataModule*) to override the default transforms. Any additional arguments are up to you.

Inside the __init__, we make a call to super. This is where we register our data sources. Data sources should be given as a dictionary which maps data source name to data source object. The name can be anything, but if you want to take advantage of our built-in from_* classmethods, you should use *DefaultDataSources* as the names. In our case, we have both a NUMPY and a custom scikit-learn data source (which we'll call *"sklearn"*).

You should also provide a default_data_source. This is the name of the data source to use by default when predicting. It'd be cool if we could get predictions just from a numpy array, so we'll use NUMPY as the default.

Here's our TemplatePreprocess.__init__:

```
def __init__(
    self,
    train_transform: Optional[Dict[str, Callable]] = None,
    val_transform: Optional[Dict[str, Callable]] = None,
    test_transform: Optional[Dict[str, Callable]] = None,
    predict_transform: Optional[Dict[str, Callable]] = None,
):
    super().__init__(
        train_transform=train_transform,
        val_transform=val_transform,
        test_transform=test_transform,
        predict_transform=predict_transform,
        data_sources={
```

(continues on next page)

(continued from previous page)

```
            DefaultDataSources.NUMPY: TemplateNumpyDataSource(),
            "sklearn": TemplateSKLearnDataSource(),
        },
        default_data_source=DefaultDataSources.NUMPY,
    )
```

For our `TemplatePreprocess`, we'll just configure a default `to_tensor_transform`. Let's first define the transform as a `staticmethod`:

```
@staticmethod
def input_to_tensor(input: np.ndarray):
    """Transform which creates a tensor from the given numpy ``ndarray`` and converts it␣
↪to ``float``"""
    return torch.from_numpy(input).float()
```

Our inputs samples will be dictionaries whose keys are in the *DefaultDataKeys*. You can map each key to different transforms using *ApplyToKeys*. Here's our `default_transforms` method:

```
def default_transforms(self) -> Optional[Dict[str, Callable]]:
    """Configures the default ``to_tensor_transform``.

    Returns:
        Our dictionary of transforms.
    """
    return {
        "to_tensor_transform": nn.Sequential(
            ApplyToKeys(DefaultDataKeys.INPUT, self.input_to_tensor),
            ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
        ),
    }
```

49.3 DataModule

The *DataModule* is responsible for creating the `DataLoader` and injecting the transforms for each stage. When the user calls a `from_*` method (such as *from_numpy()*), the following steps take place:

1. The *from_data_source()* method is called with the name of the *DataSource* to use and the inputs to provide to *load_data()* for each stage.
2. The *Preprocess* is created from `cls.preprocess_cls` (if it wasn't provided by the user) with any provided transforms.
3. The *DataSource* of the provided name is retrieved from the *Preprocess*.
4. A *BaseAutoDataset* is created from the *DataSource* for each stage.
5. The *DataModule* is instantiated with the data sets.

To create our `TemplateData` *DataModule*, we first need to attach out preprocess class like this:

```
preprocess_cls = TemplatePreprocess
```

Since we provided a NUMPY *DataSource* in the TemplatePreprocess, *from_numpy()* will now work with our TemplateData.

If you've defined a fully custom *DataSource* (like our TemplateSKLearnDataSource), then you will need to write a from_* method for each. Here's the from_sklearn method for our TemplateData:

```
@classmethod
def from_sklearn(
    cls,
    train_bunch: Optional[Bunch] = None,
    val_bunch: Optional[Bunch] = None,
    test_bunch: Optional[Bunch] = None,
    predict_bunch: Optional[Bunch] = None,
    train_transform: Optional[Dict[str, Callable]] = None,
    val_transform: Optional[Dict[str, Callable]] = None,
    test_transform: Optional[Dict[str, Callable]] = None,
    predict_transform: Optional[Dict[str, Callable]] = None,
    data_fetcher: Optional[BaseDataFetcher] = None,
    preprocess: Optional[Preprocess] = None,
    val_split: Optional[float] = None,
    batch_size: int = 4,
    num_workers: Optional[int] = None,
    **preprocess_kwargs: Any,
):
    """This is our custom ``from_*`` method. It expects scikit-learn ``Bunch`` objects as␣
↪input and passes them
    through to the :meth:`~flash.core.data.data_module.DataModule.from_data_source`␣
↪method underneath.

    Args:
        train_bunch: The scikit-learn ``Bunch`` containing the train data.
        val_bunch: The scikit-learn ``Bunch`` containing the validation data.
        test_bunch: The scikit-learn ``Bunch`` containing the test data.
        predict_bunch: The scikit-learn ``Bunch`` containing the predict data.
        train_transform: The dictionary of transforms to use during training which maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        val_transform: The dictionary of transforms to use during validation which maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        test_transform: The dictionary of transforms to use during testing which maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        predict_transform: The dictionary of transforms to use during predicting which␣
↪maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        data_fetcher: The :class:`~flash.core.data.callback.BaseDataFetcher` to pass to␣
↪the
            :class:`~flash.core.data.data_module.DataModule`.
        preprocess: The :class:`~flash.core.data.data.Preprocess` to pass to the
```

(continues on next page)

(continued from previous page)

```
            :class:`~flash.core.data.data_module.DataModule`. If ``None``, ``cls.preprocess_
↪cls`` will be
            constructed and used.
        val_split: The ``val_split`` argument to pass to the :class:`~flash.core.data.data_
↪module.DataModule`.
        batch_size: The ``batch_size`` argument to pass to the :class:`~flash.core.data.
↪data_module.DataModule`.
        num_workers: The ``num_workers`` argument to pass to the :class:`~flash.core.data.
↪data_module.DataModule`.
        preprocess_kwargs: Additional keyword arguments to use when constructing the␣
↪preprocess. Will only be used
            if ``preprocess = None``.

    Returns:
        The constructed data module.
    """
    return super().from_data_source(
        "sklearn",
        train_bunch,
        val_bunch,
        test_bunch,
        predict_bunch,
        train_transform=train_transform,
        val_transform=val_transform,
        test_transform=test_transform,
        predict_transform=predict_transform,
        data_fetcher=data_fetcher,
        preprocess=preprocess,
        val_split=val_split,
        batch_size=batch_size,
        num_workers=num_workers,
        **preprocess_kwargs,
    )
```

The final step is to implement the `num_features` property for our `TemplateData`. This is just a convenience for the user that finds the `num_features` attribute on any of the data sets and returns it. Here's the code:

```
@property
def num_features(self) -> Optional[int]:
    """Tries to get the ``num_features`` from each dataset in turn and returns the output.
↪"""
    n_fts_train = getattr(self.train_dataset, "num_features", None)
    n_fts_val = getattr(self.val_dataset, "num_features", None)
    n_fts_test = getattr(self.test_dataset, "num_features", None)
    return n_fts_train or n_fts_val or n_fts_test
```

49.4 BaseVisualization

An optional step is to implement a *BaseVisualization*. The *BaseVisualization* lets you control how data at various points in the pipeline can be visualized. This is extremely useful for debugging purposes, allowing users to view their data and understand the impact of their transforms.

Note: Don't worry about implementing it right away, you can always come back and add it later!

Here's the code for our `TemplateVisualization` which just prints the data:

```
class TemplateVisualization(BaseVisualization):
    """The ``TemplateVisualization`` class is a :class:`~flash.core.data.callbacks.
↪BaseVisualization` that just
    prints the data.

    If you want to provide a visualization with your task, you can override these hooks.
    """

    def show_load_sample(self, samples: List[Any], running_stage: RunningStage):
        print(samples)

    def show_pre_tensor_transform(self, samples: List[Any], running_stage: RunningStage):
        print(samples)

    def show_to_tensor_transform(self, samples: List[Any], running_stage: RunningStage):
        print(samples)

    def show_post_tensor_transform(self, samples: List[Any], running_stage:␣
↪RunningStage):
        print(samples)

    def show_per_batch_transform(self, batch: List[Any], running_stage):
        print(batch)
```

We can configure our custom visualization in the `TemplateData` using *configure_data_fetcher()* like this:

```
@staticmethod
def configure_data_fetcher(*args, **kwargs) -> BaseDataFetcher:
    """We can, *optionally*, provide a data visualization callback using the ``configure_
↪data_fetcher``
    method."""
    return TemplateVisualization(*args, **kwargs)
```

49.5 Postprocess

Postprocess contains any transforms that need to be applied *after* the model. You may want to use it for: converting tokens back into text, applying an inverse normalization to an output image, resizing a generated image back to the size of the input, etc. As an example, here's the `TextClassificationPostprocess` which gets the logits from a `SequenceClassifierOutput`:

```
class TextClassificationPostprocess(Postprocess):
    def per_batch_transform(self, batch: Any) -> Any:
        if isinstance(batch, SequenceClassifierOutput):
            batch = batch.logits
        return super().per_batch_transform(batch)
```

In your *DataSource* or *Preprocess*, you can add metadata to the batch using the `METADATA` key. Your *Postprocess* can then use this metadata in its transforms. You should use this approach if your postprocessing depends on the state of the input before the *Preprocess* transforms. For example, if you want to resize the predictions to the original size of the inputs you should add the original image size in the `METADATA`. Here's an example from the `SemanticSegmentationNumpyDataSource`:

```
def load_sample(self, sample: Dict[str, Any], dataset: Optional[Any] = None) -> Dict[str,
↪ Any]:
    img = torch.from_numpy(sample[DefaultDataKeys.INPUT]).float()
    sample[DefaultDataKeys.INPUT] = img
    sample[DefaultDataKeys.METADATA] = {"size": img.shape}
    return sample
```

The `METADATA` can now be referenced in your *Postprocess*. For example, here's the code for the `per_sample_transform` method of the *SemanticSegmentationPostprocess*:

```
def per_sample_transform(self, sample: Any) -> Any:
    resize = K.geometry.Resize(sample[DefaultDataKeys.METADATA]["size"][-2:],␣
↪interpolation="bilinear")
    sample[DefaultDataKeys.PREDS] = resize(sample[DefaultDataKeys.PREDS])
    sample[DefaultDataKeys.INPUT] = resize(sample[DefaultDataKeys.INPUT])
    return super().per_sample_transform(sample)
```

Now that you've got some data, it's time to *add some backbones for your task!*

CHAPTER

FIFTY

THE BACKBONES

Now that you've got a way of loading data, you should implement some backbones to use with your *Task*. Create a *FlashRegistry* to use with your *Task* in backbones.py.

The registry allows you to register backbones for your task that can be selected by the user. The backbones can come from anywhere as long as you can register a function that loads the backbone. Furthermore, the user can add their own models to the existing backbones, without having to write their own *Task*!

You can create a registry like this:

```
TEMPLATE_BACKBONES = FlashRegistry("backbones")
```

Let's add a simple MLP backbone to our registry. We need a function that creates the backbone and returns it along with the output size (so that we can create the model head in our *Task*). You can use any name for the function, although we use `load_{model name}` by convention. You also need to provide `name` and `namespace` of the backbone. The standard for *namespace* is `data_type/task_type`, so for an image classification task the namespace will be `image/classification`. Here's the code:

```
@TEMPLATE_BACKBONES(name="mlp-128", namespace="template/classification")
def load_mlp_128(num_features, **_):
    """A simple MLP backbone with 128 hidden units."""
    return (
        nn.Sequential(
            nn.Linear(num_features, 128),
            nn.ReLU(True),
            nn.BatchNorm1d(128),
        ),
        128,
    )
```

Here's another example with a slightly more complex model:

```
@TEMPLATE_BACKBONES(name="mlp-128-256", namespace="template/classification")
def load_mlp_128_256(num_features, **_):
    """An two layer MLP backbone with 128 and 256 hidden units respectively."""
    return (
        nn.Sequential(
            nn.Linear(num_features, 128),
            nn.ReLU(True),
            nn.BatchNorm1d(128),
            nn.Linear(128, 256),
            nn.ReLU(True),
            nn.BatchNorm1d(256),
```

(continues on next page)

(continued from previous page)

```
        ),
        256,
    )
```

Here's a another example, which adds `DINO` pretrained model from PyTorch Hub to the `IMAGE_CLASSIFIER_BACKBONES`, from flash/image/classification/backbones/transformers.py:

```
def dino_vitb16(*_, **__):
    backbone = torch.hub.load("facebookresearch/dino:main", "dino_vitb16")
    return backbone, 768
```

Once you've got some data and some backbones, *implement your task!*

CHAPTER

FIFTYONE

THE TASK

Once you've implemented a Flash *DataModule* and some backbones, you should implement your *Task* in model.py. The *Task* is responsible for: setting up the backbone, performing the forward pass of the model, and calculating the loss and any metrics. Remember that, under the hood, the Flash *Task* is simply a `LightningModule` with some helpful defaults.

To build your task, you can start by overriding the base *Task* or any of the existing *Task* implementations. For example, in our scikit-learn example, we can just override *ClassificationTask* which provides good defaults for classification.

You should attach your backbones registry as a class attribute like this:

```
class TemplateSKLearnClassifier(ClassificationTask):

    backbones: FlashRegistry = TEMPLATE_BACKBONES
```

51.1 Model architecture and hyper-parameters

In the `__init__()`, you will need to configure defaults for the:

- loss function
- optimizer
- metrics
- backbone / model

You will also need to create the backbone from the registry and create the model head. Here's the code:

```
def __init__(
    self,
    num_features: int,
    num_classes: int,
    backbone: Union[str, Tuple[nn.Module, int]] = "mlp-128",
    backbone_kwargs: Optional[Dict] = None,
    loss_fn: Optional[Callable] = None,
    optimizer: Union[Type[torch.optim.Optimizer], torch.optim.Optimizer] = torch.optim.
↪Adam,
    optimizer_kwargs: Optional[Dict[str, Any]] = None,
    scheduler: Optional[Union[Type[_LRScheduler], str, _LRScheduler]] = None,
    scheduler_kwargs: Optional[Dict[str, Any]] = None,
    metrics: Union[torchmetrics.Metric, Mapping, Sequence, None] = None,
```

(continues on next page)

(continued from previous page)

```
    learning_rate: float = 1e-2,
    multi_label: bool = False,
    serializer: Optional[Union[Serializer, Mapping[str, Serializer]]] = None,
):
    super().__init__(
        model=None,
        loss_fn=loss_fn,
        optimizer=optimizer,
        optimizer_kwargs=optimizer_kwargs,
        scheduler=scheduler,
        scheduler_kwargs=scheduler_kwargs,
        metrics=metrics,
        learning_rate=learning_rate,
        multi_label=multi_label,
        serializer=serializer or Labels(),
    )

    self.save_hyperparameters()

    if not backbone_kwargs:
        backbone_kwargs = {}

    if isinstance(backbone, tuple):
        self.backbone, out_features = backbone
    else:
        self.backbone, out_features = self.backbones.get(backbone)(num_features=num_
↪features, **backbone_kwargs)

    self.head = nn.Linear(out_features, num_classes)
```

Note: We call `save_hyperparameters()` to log the arguments to the `__init__` as hyperparameters. Read more here.

51.2 Adding the model routines

You should override the `{train,val,test,predict}_step` methods. The default `{train,val,test,predict}_step` implementations in *Task* expect a tuple containing the input (to be passed to the model) and target (to be used when computing the loss), and should be suitable for most applications. In our template example, we just extract the input and target from the input mapping and forward them to the `super` methods. Here's the code for the `training_step`:

```
def training_step(self, batch: Any, batch_idx: int) -> Any:
    """For the training step, we just extract the :attr:`~flash.core.data.data_source.
↪DefaultDataKeys.INPUT` and
    :attr:`~flash.core.data.data_source.DefaultDataKeys.TARGET` keys from the input and␣
↪forward them to the
    :meth:`~flash.core.model.Task.training_step`."""
    batch = (batch[DefaultDataKeys.INPUT], batch[DefaultDataKeys.TARGET])
    return super().training_step(batch, batch_idx)
```

We use the same code for the `validation_step` and `test_step`. For `predict_step` we don't need the targets, so our code looks like this:

```
def predict_step(self, batch: Any, batch_idx: int, dataloader_idx: int = 0) -> Any:
    """For the predict step, we just extract the :attr:`~flash.core.data.data_source.
↪DefaultDataKeys.INPUT` key
    from the input and forward it to the :meth:`~flash.core.model.Task.predict_step`."""
    batch = batch[DefaultDataKeys.INPUT]
    return super().predict_step(batch, batch_idx, dataloader_idx=dataloader_idx)
```

Note: You can completely replace the `{train,val,test,predict}_step` methods (that is, without a call to `super`) if you need more custom behaviour for your *`Task`* at a particular stage.

Finally, we use our backbone and head in a custom forward pass:

```
def forward(self, x) -> torch.Tensor:
    """First call the backbone, then the model head."""
    x = self.backbone(x)
    return self.head(x)
```

Now that you've got your task, take a look at some *optional advanced features you can add* or go ahead and *create some examples showing your task in action!*

CHAPTER

FIFTYTWO

OPTIONAL EXTRAS

52.1 Organize your transforms in transforms.py

If you have a lot of default transforms, it can be useful to put them all in a `transforms.py` file, to be referenced in your *Preprocess*. Here's an example from image/classification/transforms.py which creates some default transforms given the desired image size:

```
def default_transforms(image_size: Tuple[int, int]) -> Dict[str, Callable]:
    """The default transforms for image classification: resize the image, convert the␣
↪image and target to a tensor,
    collate the batch, and apply normalization."""
    if _KORNIA_AVAILABLE and os.getenv("FLASH_TESTING", "0") != "1":
        #  Better approach as all transforms are applied on tensor directly
        return {
            "to_tensor_transform": nn.Sequential(
                ApplyToKeys(DefaultDataKeys.INPUT, torchvision.transforms.ToTensor()),
                ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
            ),
            "post_tensor_transform": ApplyToKeys(
                DefaultDataKeys.INPUT,
                K.geometry.Resize(image_size),
            ),
            "collate": kornia_collate,
            "per_batch_transform_on_device": ApplyToKeys(
                DefaultDataKeys.INPUT,
                K.augmentation.Normalize(torch.tensor([0.485, 0.456, 0.406]), torch.
↪tensor([0.229, 0.224, 0.225])),
            ),
        }
    return {
        "pre_tensor_transform": ApplyToKeys(DefaultDataKeys.INPUT, T.Resize(image_size)),
        "to_tensor_transform": nn.Sequential(
            ApplyToKeys(DefaultDataKeys.INPUT, torchvision.transforms.ToTensor()),
            ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
        ),
        "post_tensor_transform": ApplyToKeys(
            DefaultDataKeys.INPUT,
            T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ),
        "collate": kornia_collate,
    }
```

Here's how we create our transforms in the *ImageClassificationPreprocess*:

```
    def default_transforms(self) -> Optional[Dict[str, Callable]]:
        return default_transforms(self.image_size)
```

52.2 Add output serializers to your Task

We recommend that you do most of the heavy lifting in the *Postprocess*. Specifically, it should include any formatting and transforms that should always be applied to the predictions. If you want to support different use cases that require different prediction formats, you should add some *Serializer* implementations in a `serialization.py` file.

Some good examples are in flash/core/classification.py. Here's the *Classes Serializer*:

```
class Classes(PredsClassificationSerializer):
    """A :class:`.Serializer` which applies an argmax to the model outputs (either logits␣
↪or probabilities) and
    converts to a list.

    Args:
        multi_label: If true, treats outputs as multi label logits.
        threshold: The threshold to use for multi_label classification.
    """

    def __init__(self, multi_label: bool = False, threshold: float = 0.5):
        super().__init__(multi_label)

        self.threshold = threshold

    def serialize(self, sample: Any) -> Union[int, List[int]]:
        sample = super().serialize(sample)
        if self.multi_label:
            one_hot = (sample.sigmoid() > self.threshold).int().tolist()
            result = []
            for index, value in enumerate(one_hot):
                if value == 1:
                    result.append(index)
            return result
        return torch.argmax(sample, -1).tolist()
```

Alternatively, here's the *Logits Serializer*:

```
class Logits(PredsClassificationSerializer):
    """A :class:`.Serializer` which simply converts the model outputs (assumed to be␣
↪logits) to a list."""

    def serialize(self, sample: Any) -> Any:
        return super().serialize(sample).tolist()
```

Take a look at *Predictions (inference)* to learn more.

Once you've added any optional extras, it's time to *create some examples showing your task in action!*

CHAPTER

FIFTYTHREE

THE EXAMPLE

Now you've implemented your task, it's time to add an example showing how cool it is! We usually provide one example in flash_examples/. You can base these off of our `template.py` examples.

The example should:

1. download the data (we'll add the example to our CI later on, so choose a dataset small enough that it runs in reasonable time)
2. load the data into a *DataModule*
3. create an instance of the *Task*
4. create a *Trainer*
5. call *finetune()* or *fit()* to train your model
6. generate predictions for a few examples
7. save the checkpoint

For our template example we don't have a pretrained backbone, so we can just call *fit()* rather than *finetune()*. Here's the full example (flash_examples/template.py):

```
import numpy as np
import torch
from sklearn import datasets

import flash
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Create the DataModule
datamodule = TemplateData.from_sklearn(
    train_bunch=datasets.load_iris(),
    val_split=0.1,
)

# 2. Build the task
model = TemplateSKLearnClassifier(num_features=datamodule.num_features, num_
↪classes=datamodule.num_classes)

# 3. Create the trainer and train the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.fit(model, datamodule=datamodule)

# 4. Classify a few examples
```

(continues on next page)

(continued from previous page)

```
predictions = model.predict(
    [
        np.array([4.9, 3.0, 1.4, 0.2]),
        np.array([6.9, 3.2, 5.7, 2.3]),
        np.array([7.2, 3.0, 5.8, 1.6]),
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("template_model.pt")
```

We get this output:

```
['setosa', 'virginica', 'versicolor']
```

Now that you've got an example showing your awesome task in action, it's time to *write some tests!*

CHAPTER

FIFTYFOUR

THE TESTS

Our next step is to create some tests for our *Task*. For the `TemplateSKLearnClassifier`, we will just create some basic tests. You should expand on these to include tests for any specific functionality you have in your *Task*.

54.1 Smoke tests

We use smoke tests, usually called `test_smoke`, throughout. These just instantiate the class we are testing, to see that they can be created without raising any errors.

54.2 tests/examples/test_scripts.py

Before we write our custom tests, we should add out examples to the CI. To do this, add a line for each example (`finetuning` and `predict`) to the annotation of `test_example` in tests/examples/test_scripts.py. Here's how those lines look for our `template.py` examples:

```
pytest.param(
    "finetuning", "template.py", marks=pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason=
↪"sklearn isn't installed")
),
...
pytest.param(
    "predict", "template.py", marks=pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason=
↪"sklearn isn't installed")
),
```

54.3 test_data.py

The most important tests in test_data.py check that the `from_*` methods work correctly. In the class `TestTemplateData`, we have two of these: `test_from_numpy` and `test_from_sklearn`. In general, there should be one `test_from_*` method for each `data_source` you have configured.

Here's the code for `test_from_numpy`:

```
    def test_from_numpy(self):
        """Tests that ``TemplateData`` is properly created when using the ``from_numpy``␣
↪method."""
        data = np.random.rand(10, self.num_features)
```

(continues on next page)

(continued from previous page)

```
        targets = np.random.randint(0, self.num_classes, (10,))

        # instantiate the data module
        dm = TemplateData.from_numpy(
            train_data=data,
            train_targets=targets,
            val_data=data,
            val_targets=targets,
            test_data=data,
            test_targets=targets,
            batch_size=2,
            num_workers=0,
        )
        assert dm is not None
        assert dm.train_dataloader() is not None
        assert dm.val_dataloader() is not None
        assert dm.test_dataloader() is not None

        # check training data
        data = next(iter(dm.train_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2,)

        # check val data
        data = next(iter(dm.val_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2,)

        # check test data
        data = next(iter(dm.test_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2,)
```

54.4 test_model.py

In test_model.py, we first have `test_forward` and `test_train`. These test that tensors can be passed to the forward and that the *Task* can be trained. Here's the code for `test_forward` and `test_train`:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
@pytest.mark.parametrize("num_classes", [4, 256])
@pytest.mark.parametrize("shape", [(1, 3), (2, 128)])
def test_forward(num_classes, shape):
    """Tests that a tensor can be given to the model forward and gives the correct␣
↪output size."""
    model = TemplateSKLearnClassifier(
        num_features=shape[1],
        num_classes=num_classes,
```

(continues on next page)

(continued from previous page)

```
    )
    model.eval()

    row = torch.rand(*shape)

    out = model(row)
    assert out.shape == (shape[0], num_classes)
```

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_train(tmpdir):
    """Tests that the model can be trained on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    train_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.fit(model, train_dl)
```

We also include tests for validating and testing: `test_val`, and `test_test`. These tests are very similar to `test_train`, but here they are for completeness:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_val(tmpdir):
    """Tests that the model can be validated on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    val_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.validate(model, val_dl)
```

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_test(tmpdir):
    """Tests that the model can be tested on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    test_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.test(model, test_dl)
```

We also include tests for prediction named `test_predict_*` for each of our data sources. In our case, we have `test_predict_numpy` and `test_predict_sklearn`. These tests should use the `data_source` argument to *`predict()`* to select the required `DataSource`. Here's `test_predict_sklearn` as an example:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_predict_sklearn():
    """Tests that we can generate predictions from a scikit-learn ``Bunch``."""
    bunch = datasets.load_iris()
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    data_pipe = DataPipeline(preprocess=TemplatePreprocess())
    out = model.predict(bunch, data_source="sklearn", data_pipeline=data_pipe)
    assert isinstance(out[0], int)
```

Now that you've written the tests, it's time to *add some docs!*

CHAPTER

FIFTYFIVE

THE DOCS

The final step is to add some docs. For each *Task* in Flash, we have a docs page in docs/source/reference. You should create a `.rst` file there with the following:

- a brief description of the task
- the predict example
- the finetuning example
- any relevant API reference

Here are the contents of docs/source/reference/template.rst which breaks down each of these steps:

```
.. _template:

########
Template
########

********
The Task
********

Here you should add a description of your task. For example:
Classification is the task of assigning one of a number of classes to each data point.

------

*******
Example
*******

.. note::

    Here you should add a short intro to your example, and then use ``literalinclude``␣
↪to add it.
    To make it simple, you can fill in this template.

Let's look at the task of <describe the task> using the <data set used in the example>.
The dataset contains <describe the data>.
Here's an outline:
```

(continues on next page)

(continued from previous page)

```
.. code-block::

    <present the folder structure of the data or some data samples here>

Once we've downloaded the data using :func:`~flash.core.data.download_data`, we create␣
↪the <link to the DataModule with ``:class:``>.
We select a pre-trained backbone to use for our <link to the Task with ``:class:``> and␣
↪finetune on the <name of the data set> data.
We then use the trained <link to the Task with ``:class:``> for inference.
Finally, we save the model.
Here's the full example:

<include the example with ``literalinclude``>

.. literalinclude:: ../../../flash_examples/template.py
    :language: python
    :lines: 14-
```

Here's the rendered doc page!

Once the docs are done, it's finally time to open a PR and wait for some reviews!

Congratulations on adding your first *Task* to Flash, we hope to see you again soon!

CHAPTER

FIFTYSIX

FLASH GOVERNANCE | PERSONS OF INTEREST

56.1 Leads

- William Falcon (williamFalcon)
- Thomas Chaton (tchaton)
- Ethan Harris (ethanwharris)

56.2 Core Maintainers

- Jirka Borovec (Borda)
- Kaushik Bokka (kaushikb11)
- Justus Schock (justusschock)
- Carlos Mocholí (carmocca)
- Sean Narenthiran (SeanNaren)
- Akihiro Nitta (akihironitta)
- Aniket Maurya (aniketmaurya)
- Ananya Harsh Jha (ananyahjha93)
- Sivaraman Karthik Rangasai (karthikrangasai)

CHAPTER

FIFTYSEVEN

CONTRIBUTING

Welcome to the PyTorch Lightning community! We're building the most advanced research platform on the planet to implement the latest, best practices that the amazing PyTorch team rolls out!

57.1 Flash Design Principles

We encourage all sorts of contributions you're interested in adding! When coding for Flash, please follow these principles.

57.1.1 Simple Internal Code

It's useful for users to look at the code and understand very quickly what's happening. Many users won't be engineers. Thus we need to value clear, simple code over condensed ninja moves. While that's super cool, this isn't the project for that :)

57.1.2 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, eventually one popular solution becomes standard practice, and everyone follows. We try to find the best way to solve a particular problem, and then force our users to use it for readability and simplicity.

When something becomes a best practice, we add it to the framework. This is usually something like bits of code in utils or in the model file that everyone keeps adding over and over again across projects. When this happens, bring that code inside the trainer and add a flag for it.

57.1.3 Backward-compatible API

We all hate updating our deep learning packages because we don't want to refactor a bunch of stuff. In Flash, we make sure every change we make which could break an API is backward compatible with good deprecation warnings.

57.1.4 Gain User Trust

As a researcher, you can't have any part of your code going wrong. So, make thorough tests to ensure that every implementation of a new trick or subtle change is correct.

57.1.5 Interoperability

PyTorch Lightning Flash is highly interoperable with PyTorch Lightning and PyTorch.

57.2 Contribution Types

We are always looking for help implementing new features or fixing bugs.

A lot of good work has already been done in project mechanics (requirements.txt, setup.py, pep8, badges, ci, etc…) so we're in a good state there thanks to all the early contributors (even pre-beta release)!

57.2.1 Bug Fixes:

1. If you find a bug please submit a GitHub issue.
    - Make sure the title explains the issue.
    - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
    - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
    - Convert your minimal code example to a unit/integration test with assert on expected results.
    - Start by debugging the issue… You can run just this particular test in your IDE and draft a fix.
    - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

***Note**, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

57.2.2 New Features:

1. Submit a GitHub issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! We recommend test driven approach to adding new features as well:
    - Write a test for the functionality you want to add.
    - Write the functional code until the test passes.
4. Add/update the relevant tests!

- This PR is a good example for adding a new metric, and this one for a new logger.

57.2.3 New Tasks:

Flash is a framework of tasks for fast prototyping, baselining, finetuning and solving business and scientific problems with deep learning. Following are general guidelines for adding new tasks.

1. Models which are standard baselines
2. Whose results are reproduced properly either by us or by authors.
3. Top models which are not SOTA but highly cited for production usage / for other uses. (E.g. Mobile BERT, MobileNets, FBNets).
4. Do not reinvent the wheel, natively support torchvision, torchtext, torchaudio models.
5. Use open source licensed models.

Please raise an issue before adding a new task. Please let us know why the particular task is important for Flash.

57.2.4 Test cases:

Want to keep Lightning Flash healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features.

Tests are written using pytest.

Have a look at sample tests here.

After you have added the respective tests, you can run the tests locally with make script:

```
make test
```

Want to add a new test case and not sure how? Talk to us!

57.3 Guidelines

For this section, we refer to read the parent PL guidelines

Reminder

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - `This code is inspired from http://...`. In case you adding new dependencies, make sure that they are compatible with the actual PyTorch Lightning license (ie. dependencies should be *at least* as permissive as the PyTorch Lightning license).

57.3.1 How to rebase my PR?

We recommend creating a PR in a separate branch other than `master`, especially if you plan to submit several changes and do not want to wait until the first one is resolved (we can work on them in parallel).

First, make sure you have set upstream by running:

```
git remote add upstream https://github.com/PyTorchLightning/lightning-flash.git
```

You'll know its set up right if you run `git remote -v` and see something similar to this:

```
origin  https://github.com/{YOUR_USERNAME}/lightning-flash.git (fetch)
origin  https://github.com/{YOUR_USERNAME}/lightning-flash.git (push)
upstream        https://github.com/PyTorchLightning/lightning-flash.git (fetch)
upstream        https://github.com/PyTorchLightning/lightning-flash.git (push)
```

Checkout your feature branch and rebase it with upstream's master before pushing up your feature branch:

```
git fetch --all --prune
git rebase upstream/master
# follow git instructions to resolve conflicts
git push -f
```

57.3.2 Question & Answer

1. **How can I help/contribute?**

   All help is extremely welcome - reporting bugs, fixing documentation, adding test cases, solving issues and preparing bug fixes. To solve some issues you can start with label good first issue or chose something close to your domain with label help wanted. Before you start to implement anything check that the issue description that it is clear and self-assign the task to you (if it is not possible, just comment that you take it and we assign it to you…).

2. **Is there a recommendation for branch names?**

   We do not rely on the name convention so far you are working with your own fork. Anyway it would be nice to follow this convention `<type>/<issue-id>_<short-name>` where the types are: `bugfix`, `feature`, `docs`, `tests`, …

3. **I have a model in other framework than PyTorch, how do I add it here?**

   Since PyTorch Lightning is written on top of PyTorch. We need models in PyTorch only. Also, we would need same or equivalent results with PyTorch Lightning after converting the models from other frameworks.

CHAPTER

FIFTYEIGHT

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog.

58.1 [0.5.0] - 2021-09-07

58.1.1 [0.5.0] - Added

- Added support for (input, target) style datasets (e.g. torchvision) to the from_datasets method (#552)
- Added support for `from_csv` and `from_data_frame` to `ImageClassificationData` (#556)
- Added SimCLR, SwAV, Barlow-twins pretrained weights for resnet50 backbone in ImageClassifier task (#560)
- Added support for Semantic Segmentation backbones and heads from `segmentation-models.pytorch` (#562)
- Added support for nesting of `Task` objects (#575)
- Added `PointCloudSegmentation` Task (#566)
- Added `PointCloudObjectDetection` Task (#600)
- Added a `GraphClassifier` task (#73)
- Added the option to pass `pretrained` as a string to `SemanticSegmentation` to change pretrained weights to load from `segmentation-models.pytorch` (#587)
- Added support for `field` parameter for loadng JSON based datasets in text tasks. (#585)
- Added `AudioClassificationData` and an example for classifying audio spectrograms (#594)
- Added a `SpeechRecognition` task for speech to text using Wav2Vec (#586)
- Added Flash Zero, a zero code command line ML platform built with flash (#611)
- Added support for `.npy` and `.npz` files to `ImageClassificationData` and `AudioClassificationData` (#651)
- Added support for `from_csv` to the `AudioClassificationData` (#651)
- Added option to pass a `resolver` to the `from_csv` and `from_pandas` methods of `ImageClassificationData`, which is used to resolve filenames given IDs (#651)
- Added integration with IceVision for the `ObjectDetector` (#608)
- Added keypoint detection task (#608)
- Added instance segmentation task (#608)

- Added Torch ORT support to Transformer based tasks (#667)
- Added support for flash zero with the `InstanceSegmentation` and `KeypointDetector` tasks (#672)
- Added support for `in_chans` argument to the flash ResNet to control the expected number of input channels (#673)
- Added a `QuestionAnswering` task for extractive question answering (#607)
- Added automatic unwrapping of IceVision prediction objects (#727)
- Added support for the `ObjectDetector` with FiftyOne (#727)
- Added support for MP3 files to the `SpeechRecognition` task with librosa (#726)
- Added support for `from_numpy` and `from_tensors` to `AudioClassificationData` (#745)

58.1.2 [0.5.0] - Changed

- Changed how pretrained flag works for loading weights for ImageClassifier task (#560)
- Removed bolts pretrained weights for SSL from ImageClassifier task (#560)
- Changed the behaviour of the `sampler` argument of the `DataModule` to take a `Sampler` type rather than instantiated object (#651)
- Changed arguments to `ObjectDetector`, use `head` instead of `model` and append `_fpn` to the backbone name instead of the `fpn` argument (#608)

58.1.3 [0.5.0] - Fixed

- Fixed a bug where serve sanity checking would not be triggered using the latest PyTorchLightning version (#493)
- Fixed a bug where train and validation metrics weren't being correctly computed (#559)
- Fixed a bug where an uncaught ValueError could be raised when checking if a module is available (#615)
- Fixed a bug where some tasks were not compatible with PyTorch 1.7 due to use of `torch.jit.isinstance` (#611)
- Fixed a bug where custom samplers would not be properly forwarded to the data loader (#651)
- Fixed a bug where it was not possible to pass no metrics to the `ImageClassifier` or `TestClassifier` (#660)
- Fixed a bug where `drop_last` would be set to True during prediction and testing (#671)
- Fixed a bug where flash was not compatible with pytorch-lightning >= 1.4.3 (#690)

58.2 [0.4.0] - 2021-06-22

58.2.1 [0.4.0] - Added

- Added integration with FiftyOne (#360)
- Added `flash.serve` (#399)
- Added support for `torch.jit` to tasks where possible and documented task JIT compatibility (#389)
- Added option to provide a `Sampler` to the `DataModule` to use when creating a `DataLoader` (#390)
- Added support for multi-label text classification and toxic comments example (#401)

- Added a sanity checking feature to flash.serve (#423)

58.2.2 [0.4.0] - Changed

- Split `backbone` argument to `SemanticSegmentation` into `backbone` and `head` arguments (#412)

58.2.3 [0.4.0] - Fixed

- Fixed a bug where the `DefaultDataKeys.METADATA` couldn't be a dict (#393)
- Fixed a bug where the `SemanticSegmentation` task would not work as expected with finetuning callbacks (#412)
- Fixed a bug where predict batches could not be visualized with `ImageClassificationData` (#438)

58.3 [0.3.2] - 2021-06-08

58.3.1 [0.3.2] - Fixed

- Fixed a bug where `flash.Trainer.from_argparse_args` + `finetune` would not work (#382)

58.4 [0.3.1] - 2021-06-08

58.4.1 [0.3.1] - Added

- Added `deeplabv3`, `lraspp`, and `unet` backbones for the `SemanticSegmentation` task (#370)

58.4.2 [0.3.1] - Changed

- Changed the installation command for extra features (#346)
- Change resize interpolation default mode to nearest (#352)

58.4.3 [0.3.1] - Deprecated

- Deprecated `SemanticSegmentation` backbone names `torchvision/fcn_resnet50` and `torchvision/fcn_resnet101`, use `fc_resnet50` and `fcn_resnet101` instead (#370)

58.4.4 [0.3.1] - Fixed

- Fixed `flash.Trainer.add_argparse_args` not adding any arguments (#343)
- Fixed a bug where the translation task wasn't decoding tokens properly (#332)
- Fixed a bug where huggingface tokenizers were sometimes being pickled (#332)
- Fixed issue with `KorniaParallelTransforms` to assure to share the random state between transforms (#351)
- Fixed a bug where using `val_split` with `overfit_batches` would give an infinite recursion (#375)
- Fixed a bug where some timm models were mistakenly given a `global_pool` argument (#377)
- Fixed `flash.Trainer.from_argparse_args` not passing arguments correctly (#380)

58.5 [0.3.0] - 2021-05-20

58.5.1 [0.3.0] - Added

- Added DataPipeline API (#188 #141 #207)
- Added timm integration (#196)
- Added BaseViz Callback (#201)
- Added backbone API (#204)
- Added support for Iterable auto dataset (#227)
- Added multi label support (#230)
- Added support for schedulers (#232)
- Added visualisation callback for image classification (#228)
- Added Video Classification task (#216)
- Added Dino backbone for image classification (#259)
- Added Data Sources API (#256 #264 #272)
- Refactor preprocess_cls to preprocess, add Serializer, add DataPipelineState (#229)
- Added Semantic Segmentation task (#239 #287 #290)
- Added Object detection prediction example (#283)
- Added Style Transfer task and accompanying finetuning and prediction examples (#262)
- Added a Template task and tutorials showing how to contribute a task to flash (#306)

58.5.2 [0.3.0] - Changed

- Rename valid_ to val_ (#197)
- Refactor preprocess_cls to preprocess, add Serializer, add DataPipelineState (#229)

58.5.3 [0.3.0] - Fixed

- Fix DataPipeline resolution in Task (#212)
- Fixed a bug where the backbone used in summarization was not correctly passed to the postprocess (#296)

58.6 [0.2.3] - 2021-04-17

58.6.1 [0.2.3] - Added

- Added TIMM integration as backbones (#196)

58.6.2 [0.2.3] - Fixed

- Fixed nltk.download (#210)

58.7 [0.2.2] - 2021-04-05

58.7.1 [0.2.2] - Changed

- Switch to use `torchmetrics` (#169)
- Better support for `optimizer` and `schedulers` (#232)
- Update lightning version to v1.2 (#133)

58.7.2 [0.2.2] - Fixed

- Fixed classification softmax (#169)
- Fixed a bug where loading from a local checkpoint that had `pretrained=True` without an internet connection would sometimes raise an error (#237)
- Don't download data if exists (#157)

58.8 [0.2.1] - 2021-3-06

58.8.1 [0.2.1] - Added

- Added `RetinaNet` & `backbones` to `ObjectDetector` Task (#121)
- Added .csv image loading utils (#116, #117, #118)

58.8.2 [0.2.1] - Changed

- Set inputs as optional (#109)

58.8.3 [0.2.1] - Fixed

- Set minimal requirements (#62)
- Fixed VGG backbone `num_features` (#154)

58.9 [0.2.0] - 2021-02-12

58.9.1 [0.2.0] - Added

- Added `ObjectDetector` Task (#56)
- Added TabNet for tabular classification (#101)
- Added support for more backbones(mobilnet, vgg, densenet, resnext) (#45)
- Added backbones for image embedding model (#63)
- Added SWAV and SimCLR models to `imageclassifier` + backbone reorg (#68)

58.9.2 [0.2.0] - Changed

- Applied transform in `FilePathDataset` (#97)
- Moved classification integration from vision root to folder (#86)

58.9.3 [0.2.0] - Fixed

- Unfreeze default number of workers in datamodule (#57)
- Fixed wrong label in `FilePathDataset` (#94)

58.9.4 [0.2.0] - Removed

- Removed `densenet161` duplicate in `DENSENET_MODELS` (#76)
- Removed redundant `num_features` arg from Classification model (#88)

58.10 [0.1.0] - 2021-02-02

58.10.1 [0.1.0] - Added

- Added flash_notebook examples (#9)
- Added `strategy` to `trainer.finetune` with `NoFreeze`, `Freeze`, `FreezeUnfreeze`, `UnfreezeMilestones` Callbacks(#39)
- Added `SummarizationData`, `SummarizationTask` and `TranslationData`, `TranslationTask` (#37)
- Added `ImageEmbedder` (#36)

CHAPTER

FIFTYNINE

TEMPLATE

59.1 The Task

Here you should add a description of your task. For example: Classification is the task of assigning one of a number of classes to each data point.

59.2 Example

Note: Here you should add a short intro to your example, and then use `literalinclude` to add it. To make it simple, you can fill in this template.

Let's look at the task of <describe the task> using the <data set used in the example>. The dataset contains <describe the data>. Here's an outline:

```
<present the folder structure of the data or some data samples here>
```

Once we've downloaded the data using `download_data()`, we create the <link to the DataModule with `:class:`>. We select a pre-trained backbone to use for our <link to the Task with `:class:`> and finetune on the <name of the data set> data. We then use the trained <link to the Task with `:class:`> for inference. Finally, we save the model. Here's the full example:

<include the example with `literalinclude`>

```
import numpy as np
import torch
from sklearn import datasets

import flash
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Create the DataModule
datamodule = TemplateData.from_sklearn(
    train_bunch=datasets.load_iris(),
    val_split=0.1,
)
```

(continues on next page)

(continued from previous page)

```
# 2. Build the task
model = TemplateSKLearnClassifier(num_features=datamodule.num_features, num_
↪classes=datamodule.num_classes)

# 3. Create the trainer and train the model
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.device_count())
trainer.fit(model, datamodule=datamodule)

# 4. Classify a few examples
predictions = model.predict(
    [
        np.array([4.9, 3.0, 1.4, 0.2]),
        np.array([6.9, 3.2, 5.7, 2.3]),
        np.array([7.2, 3.0, 5.8, 1.6]),
    ]
)
print(predictions)

# 5. Save the model!
trainer.save_checkpoint("template_model.pt")
```

CHAPTER

SIXTY

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

F

Q

R

S

T

U

V