Flash

PyTorch Lightning

Apr 17, 2021

GET 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 advnaced 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 create more complex and custmoized 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 unecessary 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 essentialy 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 diffrent 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 Install

You can install flash using pip or conda:

```
pip install lightning-flash -U
```

1.3 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.vision 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 and existing task in just a few lines.

1.4 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!",
])

# Expect [0,0, 1] which means [negative, negative, positive]
print(predictions)
```

1.5 Finetune

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

1. Download and set up your own data (`DataLoader` or LightningModule work).
2. Init your task.
3. Init a *flash.core.trainer.Trainer* (or a Lightning Trainer).
4. Call *flash.core.trainer.Trainer.finetune()* with your data set.
5. Use your finetuned model for predictions

Here's an example of finetuning.

```
import flash
from flash import download_data
from flash.vision import ImageClassificationData, ImageClassifier

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

# 2. Load the data from folders
datamodule = ImageClassificationData.from_folders(
    backbone="resnet18",
    train_folder="data/hymenoptera_data/train/",
    valid_folder="data/hymenoptera_data/val/",
    test_folder="data/hymenoptera_data/test/",
)

# 3. Build the model using desired Task
model = ImageClassifier(num_classes=datamodule.num_classes)

# 4. Create the trainer (run one epoch for demo)
trainer = flash.Trainer(max_epochs=1)

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

# 6. Use the model for predictions
predictions = model.predict('data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg')
# Expact 1 -> bee
print(predictions)

predictions = model.predict('data/hymenoptera_data/val/ants/2255445811_dabcdf7258.jpg
↪')
# Expact 0 -> ant
print(predictions)

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

Once your model is finetuned, use it for prediction anywhere you want!

```
from flash.vision import ImageClassifier

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

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

1.6 Train

When you have enough data, you're likely better off training from scratch instead of finetuning. Steps here are similar to finetune:

1. Download and set up your own data (`DataLoader` or LightningModule work).
2. Init your task.
3. Init a *`flash.core.trainer.Trainer`* (or a Lightning Trainer).
4. Call *`flash.core.trainer.Trainer.fit()`* with your data set.
5. Use your finetuned model for predictions

1.7 A few Built-in Tasks

- *Generic Flash Task*
- *ImageClassification*
- *ImageEmbedder*
- *TextClassification*
- *SummarizationTask*
- *TranslationTask*
- *TabularClassification*

More tasks coming soon!

1.7.1 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/conda

```
pip install lightning-flash -U
```

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*.

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

```
from typing import Any, List, Tuple

import numpy as np
import torch
from pytorch_lightning import seed_everything
from sklearn import datasets
from sklearn.model_selection import train_test_split
from torch import nn

import flash
from flash.data.auto_dataset import AutoDataset
from flash.data.process import Postprocess, Preprocess

# set the random seeds.
seed_everything(42)
```

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 only need to override the `__init__` and `forward` methods.

```
class RegressionTask(flash.Task):

    def __init__(self, num_inputs, learning_rate=0.001, metrics=None):
        # what kind of model do we want?
        model = nn.Linear(num_inputs, 1)

        # what loss function do we want?
```

(continues on next page)

(continued from previous page)

```
        loss_fn = torch.nn.functional.mse_loss

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

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

    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 to learn more *Available Registries*.

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.

Example:

```
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 DataModule API

Now that we have defined our `RegressionTask`, we need to load our data. We will define a custom `NumpyDataModule` class subclassing *DataModule*. This `NumpyDataModule` class will provide a `from_xy_dataset` helper `classmethod` to instantiate *DataModule* from x, y numpy arrays.

Here is how it would look like:

Example:

```
x, y = ...
preprocess_cls = ...
datamodule = NumpyDataModule.from_xy_dataset(x, y, preprocess_cls)
```

Here is the `NumpyDataModule` implementation:

Example:

```
from flash import DataModule
from flash.data.process import Preprocess
import numpy as np

ND = np.ndarray

class NumpyDataModule(DataModule):

    @classmethod
    def from_xy_dataset(
        cls,
        x: ND,
        y: ND,
        preprocess_cls: Preprocess = NumpyPreprocess,
        batch_size: int = 64,
        num_workers: int = 0
    ):

        preprocess = preprocess_cls()

        x_train, x_test, y_train, y_test = train_test_split(
            x, y, test_size=.20, random_state=0)

        # Make sure to call ``from_load_data_inputs``.
        # The ``train_load_data_input`` value will be given to ``Preprocess``
        # ``train_load_data`` function.
        dm = cls.from_load_data_inputs(
            train_load_data_input=(x_train, y_train),
            test_load_data_input=(x_test, y_test),
            preprocess=preprocess,  # DON'T FORGET TO PROVIDE THE PREPROCESS
            batch_size=batch_size,
            num_workers=num_workers
        )
        # Some metatada can be accessed from ``train_ds`` directly.
        dm.num_inputs = dm.train_dataset.num_inputs
        return dm
```

Note: The *DataModule* provides a `from_load_data_inputs` helper function. This function will take care of connecting the provided *Preprocess* with the *DataModule*. Make sure to instantiate your *DataModule* with this helper if you rely on *Preprocess* objects.

3.4 3.b The Preprocess API

A *Preprocess* object provides a series of hooks that can be overridden with custom data processing logic. It allows the user much more granular control over their data processing flow.

Note: Why introducing *Preprocess* ?

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

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

Example:

```
import torch
from torch import Tensor
import numpy as np

ND = np.ndarray

class NumpyPreprocess(Preprocess):

    def load_data(self, data: Tuple[ND, ND], dataset: AutoDataset) -> List[Tuple[ND,␣
↪float]]:
        if self.training:
            dataset.num_inputs = data[0].shape[1]
        return [(x, y) for x, y in zip(*data)]

    def to_tensor_transform(self, sample: Any) -> Tuple[Tensor, Tensor]:
        x, y = sample
        x = torch.from_numpy(x).float()
        y = torch.tensor(y, dtype=torch.float)
        return x, y

    def predict_load_data(self, data: ND) -> ND:
        return data

    def predict_to_tensor_transform(self, sample: ND) -> ND:
        return torch.from_numpy(sample).float()
```

You now have a new customized Flash Task! Congratulations !

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

3.5 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_xy_dataset(x, y)
model = RegressionTask(num_inputs=datamodule.num_inputs)
```

(continues on next page)

(continued from previous page)

```
trainer = flash.Trainer(max_epochs=1000)
trainer.fit(model, datamodule=datamodule)
```

3.6 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 = torch.tensor([
    [ 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)
#out: [tensor([14.7190]), tensor([14.7100]), tensor([14.7288]), tensor([14.6685]),␣
↪tensor([14.6687])]
```

CHAPTER

FOUR

FROM FLASH TO LIGHTNING

Flash is built on top of Pytorch Lightning to abstract away the unecessary 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

GENERAL TASK

A majority of data science problems that involve machine learning can be tackled using Task. With Task you can:

- Pass an arbitrary model
- Pass an arbitrary loss
- Pass an arbitrary optimizer

5.1 Example: Image Classification

```
from flash import Task
from torch import nn, optim
from torch.utils.data import DataLoader, random_split
from torchvision import transforms, datasets
import pytorch_lightning as pl

# model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

# data
dataset = datasets.MNIST('./data_folder', download=True, transform=transforms.
↪ToTensor())
train, val = random_split(dataset, [55000, 5000])

# task
classifier = Task(model, loss_fn=nn.functional.cross_entropy, optimizer=optim.Adam)

# train
pl.Trainer().fit(classifier, DataLoader(train), DataLoader(val))
```

5.2 API reference

5.2.1 Task

class flash.core.**Task**(*model=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=5e-05*, *default_preprocess=None*, *default_postprocess=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** (Type[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*.
- **default_preprocess** (Optional[*Preprocess*]) – *Preprocess* to use as the default for this task.
- **default_postprocess** (Optional[*Postprocess*]) – *Postprocess* to use as the default for this task.

build_data_pipeline(*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*.

property data_pipeline

The current *DataPipeline*. If set, the new value will override the *Task* defaults. See build_data_pipeline() for more details on the resolution order.

Return type *DataPipeline*

predict(*x*, *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*)

The training/validation/test step. Override for custom behavior.

Return type Any

CHAPTER

SIX

IMAGE CLASSIFICATION

6.1 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 desecribes what is in an image, such as 'car', 'house', 'cat' etc. For example, we can train the image classifier task on images of ants and it will learn to predict the probability that an image contains an ant.

6.2 Inference

The *ImageClassifier* is already pre-trained on ImageNet, a dataset of over 14 million images.

Use the *ImageClassifier* pretrained model for inference on any string sequence using `predict()`:

```
# import our libraries
from flash import Trainer
from flash import download_data
from flash.vision import ImageClassificationData, ImageClassifier

# 1. Download the data
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"
)

# 3a. 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)

# 3b. Or generate predictions with a whole folder!
datamodule = ImageClassificationData.from_folders(predict_folder="data/hymenoptera_
↪data/predict/")
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

For more advanced inference options, see *Predictions (inference)*.

6.3 Finetuning

Lets say you wanted to develope a model that could determine whether an image contains **ants** or **bees**, using the hymenoptera dataset. Once we download the data using `download_data()`, all we need is the train data and validation data folders to create the *`ImageClassificationData`*.

Note: 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
        ...
```

Now all we need is three lines of code to build to train our task!

```
import flash
from flash import download_data
from flash.vision import ImageClassificationData, ImageClassifier

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

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

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

# 4. Create the trainer. Run once on data
trainer = flash.Trainer(max_epochs=1)
```

(continues on next page)

(continued from previous page)

```
# 5. Train the model
trainer.finetune(model, datamodule=datamodule, strategy="freeze_unfreeze")

# 6. Test the model
trainer.test()

# 7. Save it!
trainer.save_checkpoint("image_classification_model.pt")
```

6.4 Changing the backbone

By default, we use a ResNet-18 for image classification. You can change the model run by the task by passing in a different backbone.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the Data object!

```
# 1. organize the data
data = ImageClassificationData.from_folders(
    backbone="resnet34",
    train_folder="data/hymenoptera_data/train/",
    valid_folder="data/hymenoptera_data/val/"
)

# 2. build the task
task = ImageClassifier(num_classes=2, backbone="resnet34")
```

Available backbones:

- resnet18 (default)
- resnet34
- resnet50
- resnet101
- resnet152
- resnext50_32x4d
- resnext101_32x8d
- mobilenet_v2
- vgg11
- vgg13
- vgg16
- vgg19
- densenet121
- densenet169

- densenet161
- swav-imagenet

6.5 API reference

6.5.1 ImageClassifier

class flash.vision.**ImageClassifier**(*num_classes*, *backbone='resnet18'*, *backbone_kwargs=None*, *head=None*, *pretrained=True*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.SGD*, *metrics=torchmetrics.Accuracy*, *learning_rate=0.001*)

Task that classifies images.

Use a built in backbone

Example:

```
from flash.vision import ImageClassifier

classifier = ImageClassifier(backbone='resnet18')
```

Or your own backbone (num_features is the number of features produced by your backbone)

Example:

```
from flash.vision import ImageClassifier
from torch import nn

# use any backbone
some_backbone = nn.Conv2D(...)
num_out_features = 1024
classifier = ImageClassifier(backbone=(some_backbone, num_out_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** (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.
- **metrics** (Union[Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation, defaults to torchmetrics.Accuracy.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.

6.5.2 ImageClassificationData

class flash.vision.**ImageClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=None*, *seed=1234*, *train_split=None*, *val_split=None*, *test_split=None*, ***kwargs*)

Data module for image classification tasks.

classmethod ImageClassificationData.**from_filepaths**(*train_filepaths=None*, *train_labels=None*, *val_filepaths=None*, *val_labels=None*, *test_filepaths=None*, *test_labels=None*, *predict_filepaths=None*, *train_transform='default'*, *val_transform='default'*, *test_transform='default'*, *predict_transform='default'*, *batch_size=64*, *num_workers=None*, *seed=42*, *preprocess_cls=None*, ***kwargs*)

Creates a ImageClassificationData object from folders of images arranged in this way:

```
folder/dog_xxx.png
folder/dog_xxy.png
folder/dog_xxz.png
folder/cat_123.png
folder/cat_nsdf3.png
folder/cat_asd932_.png
```

Parameters

- **train_filepaths** (Union[str, Path, Sequence[Union[str, Path]], None]) – String or sequence of file paths for training dataset. Defaults to None.
- **train_labels** (Optional[Sequence]) – Sequence of labels for training dataset. Defaults to None.
- **val_filepaths** (Union[str, Path, Sequence[Union[str, Path]], None]) – String or sequence of file paths for validation dataset. Defaults to None.
- **val_labels** (Optional[Sequence]) – Sequence of labels for validation dataset. Defaults to None.
- **test_filepaths** (Union[str, Path, Sequence[Union[str, Path]], None]) – String or sequence of file paths for test dataset. Defaults to None.
- **test_labels** (Optional[Sequence]) – Sequence of labels for test dataset. Defaults to None.
- **train_transform** (Union[str, Dict]) – Transforms for training dataset. Defaults to default, which loads imagenet transforms.
- **val_transform** (Union[str, Dict]) – Transforms for validation and testing dataset. Defaults to default, which loads imagenet transforms.
- **batch_size** (int) – The batchsize to use for parallel loading. Defaults to 64.

- **num_workers** (Optional[int]) – The number of workers to use for parallelized loading. Defaults to None which equals the number of available CPU threads.
- **seed** (Optional[int]) – Used for the train/val splits.

Returns The constructed data module.

Return type *ImageClassificationData*

classmethod ImageClassificationData.**from_folders**(*train_folder=None*, *val_folder=None*, *test_folder=None*, *predict_folder=None*, *train_transform='default'*, *val_transform='default'*, *test_transform='default'*, *predict_transform='default'*, *batch_size=4*, *num_workers=None*, *preprocess_cls=None*, ***kwargs*)

Creates a ImageClassificationData object from folders of images arranged in this way:

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

Parameters

- **train_folder** (Union[str, Path, None]) – Path to training folder. Default: None.
- **val_folder** (Union[str, Path, None]) – Path to validation folder. Default: None.
- **test_folder** (Union[str, Path, None]) – Path to test folder. Default: None.
- **predict_folder** (Union[str, Path, None]) – Path to predict folder. Default: None.
- **val_transform** (Union[str, Dict, None]) – Image transform to use for validation and test set.
- **train_transform** (Union[str, Dict, None]) – Image transform to use for training set.
- **val_transform** – Image transform to use for validation set.
- **test_transform** (Union[str, Dict, None]) – Image transform to use for test set.
- **predict_transform** (Union[str, Dict, None]) – Image transform to use for predict set.
- **batch_size** (int) – Batch size for data loading.
- **num_workers** (Optional[int]) – The number of workers to use for parallelized loading. Defaults to None which equals the number of available CPU threads.

Returns the constructed data module

Return type *ImageClassificationData*

Examples

```
>>> img_data = ImageClassificationData.from_folders("train/")
```

CHAPTER

SEVEN

IMAGE EMBEDDER

7.1 The task

Image embedding encodes an image into a vector of image features which can be used for anything like clustering, similarity search or classification.

7.2 Inference

The *ImageEmbedder* is already pre-trained on ImageNet, a dataset of over 14 million images.

Use the *ImageEmbedder* pretrained model for inference on any image tensor or image path using `predict()`:

```
from flash.vision import ImageEmbedder

# Load finetuned task
embedder = ImageEmbedder(backbone="resnet18")

# 2. Perform inference on an image file
embeddings = embedder.predict("path/to/image.png")
print(embeddings)
```

Or on a random image tensor

```
# 2. Perform inference on a random image tensor
import torch
images = torch.rand(32, 3, 224, 224)
embeddings = embedder.predict(images)
print(embeddings)
```

For more advanced inference options, see *Predictions (inference)*.

7.3 Finetuning

To tailor this image embedder to your dataset, finetune first.

```
import flash
from flash import download_data
from flash.vision import ImageClassificationData, ImageEmbedder

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

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

# 3. Build the model
embedder = ImageEmbedder(backbone="resnet18", embedding_dim=128)

# 4. Create the trainer. Run once on data
trainer = flash.Trainer(max_epochs=1)

# 5. Train the model
trainer.finetune(embedder, datamodule=datamodule, strategy="freeze_unfreeze")

# 6. Test the model
trainer.test()

# 7. Save it!
trainer.save_checkpoint("image_embedder_model.pt")
```

7.4 Changing the backbone

By default, we use the encoder from SwAV pretrained on Imagenet via contrastive learning. You can change the model run by the task by passing in a different backbone.

```
# 1. organize the data
data = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    valid_folder="data/hymenoptera_data/val/"
)

# 2. build the task
embedder = ImageEmbedder(backbone="resnet34")
```

Backbones available

Table 1: Backbones

backbone	dataset	training method
resnet18	Imagenet	supervised
resnet34	Imagenet	supervised
resnet50	Imagenet	supervised
resnet101	Imagenet	supervised
resnet152	Imagenet	supervised
swav-imagenet	Imagenet	self-supervised (clustering)

7.5 API reference

7.5.1 ImageEmbedder

class flash.vision.**ImageEmbedder**(*embedding_dim=None*, *backbone='swav-imagenet'*, *pretrained=True*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.SGD*, *metrics=torchmetrics.Accuracy*, *learning_rate=0.001*, *pooling_fn=torch.max*)

Task that classifies images.

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.
- **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.
- **pooling_fn** (Callable) – Function used to pool image to generate embeddings, defaults to torch.max().

Example

```
>>> import torch
>>> from flash.vision.embedding import ImageEmbedder
>>> embedder = ImageEmbedder(backbone='resnet18')
>>> image = torch.rand(32, 3, 32, 32)
>>> embeddings = embedder(image)
```

CHAPTER

EIGHT

SUMMARIZATION

8.1 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 requires 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.

8.2 Inference

The *SummarizationTask* is already pre-trained on XSUM, a dataset of online British Broadcasting Corporation articles.

Use the *SummarizationTask* pretrained model for inference on any string sequence using *SummarizationTask* *predict* method:

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

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

# 2. Perform inference from a sequence
predictions = model.predict([
    """
    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."
↪"
```

(continues on next page)

(continued from previous page)

```
    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."
    """
])
print(predictions)
```

Or on a given dataset, use `Trainer` *predict* method:

```
# import our libraries
from flash import Trainer
from flash import download_data
from flash.text import SummarizationData, SummarizationTask

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

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

# 3. Create dataset from file
datamodule = SummarizationData.from_file(
    predict_file="data/xsum/predict.csv",
    input="input",
)

# 4. generate summaries
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

For more advanced inference options, see *Predictions (inference)*.

8.3 Finetuning

Say you want to finetune to your own summarization data. We use the XSUM dataset as an example which contains a `train.csv` and `valid.csv`, structured like so:

```
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.

All we need is three lines of code to train our model!

```
# import our libraries
import flash
from flash import download_data
from flash.text import SummarizationData, SummarizationTask

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

# Organize the data
datamodule = SummarizationData.from_files(
    train_file="data/xsum/train.csv",
    valid_file="data/xsum/valid.csv",
    test_file="data/xsum/test.csv",
    input="input",
    target="target"
)

# 2. Build the task
model = SummarizationTask()

# 4. Create trainer
trainer = flash.Trainer(max_epochs=1, gpus=1)

# 5. Finetune the task
trainer.finetune(model, datamodule=datamodule)

# 6. Save trainer task
trainer.save_checkpoint("summarization_model_xsum.pt")
```

To run the example:

```
python flash_examples/finetuning/summarization.py
```

8.4 Changing the backbone

By default, we use the t5 model for summarization. You can change the model run by the task to any summarization model from HuggingFace/transformers by passing in a backbone parameter.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the Data object! Since this is a Seq2Seq task, make sure you use a Seq2Seq model.

```
# use google/mt5-small, covering 101 languages
datamodule = SummarizationData.from_files(
    backbone="google/mt5-small",
    train_file="data/wmt_en_ro/train.csv",
    valid_file="data/wmt_en_ro/valid.csv",
    test_file="data/wmt_en_ro/test.csv",
    input="input",
    target="target",
)

model = SummarizationTask(backbone="google/mt5-small")
```

8.5 API reference

8.5.1 SummarizationTask

class flash.text.**SummarizationTask**(*backbone='t5-small'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=5e-05*, *val_target_max_length=None*, *num_beams=4*, *use_stemmer=True*, *rouge_newline_sep=True*)

Task for Seq2Seq Summarization.

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*.
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **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.

property task

Override to define AutoConfig task specific parameters stored within the model.

Return type str

8.5.2 SummarizationData

class flash.text.**SummarizationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

classmethod SummarizationData.**from_files**(*train_file=None*, *input='input'*, *target=None*, *filetype='csv'*, *backbone='t5-small'*, *val_file=None*, *test_file=None*, *predict_file=None*, *max_source_length=512*, *max_target_length=128*, *padding='max_length'*, *batch_size=16*, *num_workers=None*, *preprocess_cls=None*, *postprocess_cls=None*)

Creates a SummarizationData object from files.

Parameters

- **train_file** (Optional[str]) – Path to training data.
- **input** (str) – The field storing the source translation text.
- **target** (Optional[str]) – The field storing the target translation text.
- **filetype** (str) – .csv or .json
- **backbone** (str) – Tokenizer backbone to use, can use any HuggingFace tokenizer.
- **val_file** (Optional[str]) – Path to validation data.
- **test_file** (Optional[str]) – Path to test data.
- **max_source_length** (int) – Maximum length of the source text. Any text longer will be truncated.
- **max_target_length** (int) – Maximum length of the target text. Any text longer will be truncated.
- **padding** (Union[str, bool]) – Padding strategy for batches. Default is pad to maximum length.
- **batch_size** (int) – The batchsize to use for parallel loading. Defaults to 16.
- **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 Darwin platform.

Returns The constructed data module.

Return type *SummarizationData*

Examples:

```
train_df = pd.read_csv("train_data.csv")
tab_data = TabularData.from_df(train_df, target="fraud",
                               num_cols=["account_value"],
                               cat_cols=["account_type"])
```

CHAPTER

NINE

TEXT CLASSIFICATION

9.1 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 dataset and can range from topics. For example, we can use text classification to understand the sentiment of a given sentence- if it is positive or negative.

9.2 Inference

The *TextClassifier* is already pre-trained on IMDB, a dataset of highly polarized movie reviews, trained for binary classification- to predict if a given review has a positive or negative sentiment.

Use the *TextClassifier* pretrained model for inference on any string sequence using `predict()`:

```
from pytorch_lightning import Trainer

from flash import download_data
from flash.text import TextClassificationData, TextClassifier


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

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

# 2a. 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."
    "Very, very afraid"
    "This guy has done a great job with this movie!",
])
print(predictions)

# 2b. Or generate predictions from a sheet file!
datamodule = TextClassificationData.from_file(
    predict_file="data/imdb/predict.csv",
```

(continues on next page)

(continued from previous page)

```
    input="review",
)
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

For more advanced inference options, see *Predictions (inference)*.

9.3 Finetuning

Say you wanted to create a model that can predict whether a movie review is **positive** or **negative**. We will be using the IMDB dataset, that contains a `train.csv` and `valid.csv`, structured like so:

```
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
...
```

All we need is three lines of code to train our model!

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

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

# 2. Load the data
datamodule = TextClassificationData.from_files(
    train_file="data/imdb/train.csv",
    valid_file="data/imdb/valid.csv",
    test_file="data/imdb/test.csv",
    input="review",
    target="sentiment",
    batch_size=512
)

# 3. Build the task (using the default backbone="bert-base-cased")
model = TextClassifier(num_classes=datamodule.num_classes)

# 4. Create the trainer. Run once on data
trainer = flash.Trainer(max_epochs=1)

# 5. Finetune the task
trainer.finetune(model, datamodule=datamodule, strategy="freeze_unfreeze")

# 6. Test model
trainer.test()

# 7. Save it!
trainer.save_checkpoint("text_classification_model.pt")
```

To run the example:

```
python flash_examples/finetuning/text_classification.py
```

9.4 Changing the backbone

By default, we use the bert-base-uncased model for text classification. You can change the model run by the task to any BERT model from HuggingFace/transformers by passing in a different backbone.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the Data object!

```
datamodule = TextClassificationData.from_files(
    backbone="bert-base-chinese",
    train_file="data/imdb/train.csv",
    valid_file="data/imdb/valid.csv",
    input="review",
    target="sentiment",
    batch_size=512
)

task = TextClassifier(backbone="bert-base-chinese", num_classes=datamodule.num_
↪classes)
```

9.5 API reference

9.5.1 TextClassifier

class flash.text.classification.model.**TextClassifier**(*num_classes*, *backbone='prajjwal1/bert-tiny'*, *optimizer=torch.optim.Adam*, *metrics=[torchmetrics.Accuracy]*, *learning_rate=0.001*)

Task that classifies text.

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*.
- **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*

step(*batch*, *batch_idx*)
The training/validation/test step. Override for custom behavior.

Return type dict

9.5.2 TextClassificationData

class flash.text.classification.data.**TextClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

Data Module for text classification tasks

classmethod TextClassificationData.**from_files**(*train_file*, *input='input'*, *target='labels'*, *filetype='csv'*, *backbone='prajjwal1/bert-tiny'*, *val_file=None*, *test_file=None*, *predict_file=None*, *max_length=128*, *label_to_class_mapping=None*, *batch_size=16*, *num_workers=None*, *preprocess_state=None*, *preprocess_cls=None*)

Creates a TextClassificationData object from files.

Parameters

- **train_file**¶ (Optional[str]) – Path to training data.
- **input**¶ (Optional[str]) – The field storing the text to be classified.
- **target**¶ (Optional[str]) – The field storing the class id of the associated text.
- **filetype**¶ (str) – .csv or .json
- **backbone**¶ (str) – Tokenizer backbone to use, can use any HuggingFace tokenizer.
- **val_file**¶ (Optional[str]) – Path to validation data.
- **test_file**¶ (Optional[str]) – Path to test data.
- **batch_size**¶ (int) – the batchsize to use for parallel loading. Defaults to 64.
- **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 Darwin platform.

Returns The constructed data module.

Return type *TextClassificationData*

Examples:

```
train_df = pd.read_csv("train_data.csv")
tab_data = TabularData.from_df(train_df, target="fraud",
                               num_cols=["account_value"],
                               cat_cols=["account_type"])
```

CHAPTER

TEN

TABULAR CLASSIFICATION

10.1 The task

Tabular classification is the task of assigning a class to samples of structured or relational data. The Flash Tabular Classification task can be used for multi-class classification, or classification of samples in more than two classes. In the following example, the Tabular data is structured into rows and columns, where columns represent properties or features. The task will learn to predict a single target column.

10.2 Finetuning

Say we want to build a model to predict if a passenger survived on the Titanic. We can organize our data in `.csv` files (exportable from Excel, but you can find the kaggle dataset here):

```
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
...
```

We can use the Flash Tabular classification task to predict the probability a passenger survived (1 means survived, 0 otherwise), using the feature columns.

We can create *`TabularData`* from csv files using the *`from_csv()`* method. We will pass in:

- **train_csv**- csv file containing the training data converted to a Pandas DataFrame
- **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

Next, we create the *`TabularClassifier`* task, using the Data module we created.

```
import flash
from flash import download_data
from flash.tabular import TabularClassifier, TabularData
from torchmetrics.classification import Accuracy, Precision, Recall

# 1. Download the data
```

(continues on next page)

(continued from previous page)

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

# 2. Load the data
datamodule = TabularData.from_csv(
    "./data/titanic/titanic.csv",
    test_csv="./data/titanic/test.csv",
    cat_cols=["Sex", "Age", "SibSp", "Parch", "Ticket", "Cabin", "Embarked"],
    num_cols=["Fare"],
    target="Survived",
    val_size=0.25,
    )

# 3. Build the model
model = TabularClassifier.from_data(datamodule, metrics=[Accuracy(), Precision(),␣
↪Recall()])

# 4. Create the trainer. Run 10 times on data
trainer = flash.Trainer(max_epochs=10)

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

# 6. Test model
trainer.test()

# 7. Save it!
trainer.save_checkpoint("tabular_classification_model.pt")

# 8. Predict!
predictions = model.predict("data/titanic/titanic.csv")
print(predictions)
```

10.3 Inference

You can make predcitions on a pretrained model, that has already been trained for the titanic task:

```
from flash.core.data 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 sheet file! Who would survive?
predictions = model.predict("data/titanic/titanic.csv")
print(predictions)
```

Or you can finetune your own model and use that for prediction:

```
import flash
from flash import download_data
from flash.tabular import TabularClassifier, TabularData

# 1. Load the data
datamodule = TabularData.from_csv(
    "my_data_file.csv",
    test_csv="./data/titanic/test.csv",
    cat_cols=["Sex", "Age", "SibSp", "Parch", "Ticket", "Cabin", "Embarked"],
    num_cols=["Fare"],
    target="Survived",
    val_size=0.25,
)

# 3. Build the model
model = TabularClassifier.from_data(datamodule, metrics=[Accuracy(), Precision(),␣
↪Recall()])

# 4. Create the trainer
trainer = flash.Trainer()

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

# 6. Test model
trainer.test()

predictions = model.predict("data/titanic/titanic.csv")
print(predictions)
```

10.4 API reference

10.4.1 TabularClassifier

class flash.tabular.**TabularClassifier**(*num_features*, *num_classes*, *embedding_sizes=None*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=0.001*, ***tabnet_kwargs*)

Task that classifies table rows.

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]]) – 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** (Optional[List[Metric]]) – Metrics to compute for training and evaluation.

- **learning_rate** (float) – Learning rate to use for training, defaults to *1e-3*
- ****tabnet_kwargs** – Optional additional arguments for the TabNet model, see pytorch_tabnet.

10.4.2 TabularData

class flash.tabular.**TabularData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

Data module for tabular tasks

classmethod TabularData.**from_csv**(*target_col*, *train_csv=None*, *categorical_cols=None*, *numerical_cols=None*, *val_csv=None*, *test_csv=None*, *predict_csv=None*, *batch_size=8*, *num_workers=None*, *val_size=None*, *test_size=None*, *preprocess_cls=None*, *preprocess_state=None*, ***pandas_kwargs*)

Creates a TextClassificationData object from pandas DataFrames.

Parameters

- **train_csv** (Optional[str]) – Train data csv file.
- **target_col** (str) – The column containing the class id.
- **categorical_cols** (Optional[List]) – The list of categorical columns.
- **numerical_cols** (Optional[List]) – The list of numerical columns.
- **val_csv** (Optional[str]) – Validation data csv file.
- **test_csv** (Optional[str]) – Test data csv file.
- **batch_size** (int) – The batchsize to use for parallel loading. Defaults to 64.
- **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 Darwin platform.
- **val_size** (Optional[float]) – Float between 0 and 1 to create a validation dataset from train dataset.
- **test_size** (Optional[float]) – Float between 0 and 1 to create a test dataset from train validation.
- **preprocess_cls** (Optional[Type[*Preprocess*]]) – Preprocess class to be used within this DataModule DataPipeline.
- **preprocess_state** (Optional[TabularState]) – Used to store the train statistics.

Returns The constructed data module.

Return type *TabularData*

Examples:

```
text_data = TabularData.from_files("train.csv", label_field="class", text_field=
↪"sentence")
```

classmethod TabularData.**from_df**(*train_df*, *target_col*, *categorical_cols=None*, *numerical_cols=None*, *val_df=None*, *test_df=None*, *predict_df=None*, *batch_size=8*, *num_workers=None*, *val_size=None*, *test_size=None*, *is_regression=False*, *preprocess_state=None*, *preprocess_cls=None*)

Creates a TabularData object from pandas DataFrames.

Parameters

- **train_df** (DataFrame) – Train data DataFrame.
- **target_col** (str) – The column containing the class id.
- **categorical_cols** (Optional[List]) – The list of categorical columns.
- **numerical_cols** (Optional[List]) – The list of numerical columns.
- **val_df** (Optional[DataFrame]) – Validation data DataFrame.
- **test_df** (Optional[DataFrame]) – Test data DataFrame.
- **batch_size** (int) – The batchsize to use for parallel loading. Defaults to 64.
- **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 Darwin platform.
- **val_size** (Optional[float]) – Float between 0 and 1 to create a validation dataset from train dataset.
- **test_size** (Optional[float]) – Float between 0 and 1 to create a test dataset from train validation.

Returns The constructed data module.

Return type *TabularData*

Examples:

```
text_data = TextClassificationData.from_files("train.csv", label_field="class",␣
↪text_field="sentence")
```

CHAPTER

ELEVEN

TRANSLATION

11.1 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.

11.2 Inference

The *TranslationTask* is already pre-trained on WMT16 English/Romanian, a dataset of English to Romanian samples, based on the Europarl corpora.

Use the *TranslationTask* pretrained model for inference on any string sequence using *TranslationTask* *predict* method:

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

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

# 2. Perform inference from list of sequences
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.",
])
print(predictions)
```

Or on a given dataset, use *Trainer* *predict* method:

```
# import our libraries
from flash import Trainer
from flash import download_data
from flash.text import TranslationData, TranslationTask

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

(continues on next page)

(continued from previous page)

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

# 3. Create dataset from file
datamodule = TranslationData.from_file(
    predict_file="data/wmt_en_ro/predict.csv",
    input="input",
)

# 4. generate translations
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

For more advanced inference options, see *Predictions (inference)*.

11.3 Finetuning

Say you want to finetune to your own translation data. We use the English/Romanian WMT16 dataset as an example which contains a `train.csv` and `valid.csv`, structured like so:

```
input,target
"Written statements and oral questions (tabling): see Minutes","Declaraţii scrise şi␣
↪întrebări orale (depunere): consultaţi procesul-verbal"
"Closure of sitting","Ridicarea şedinţei"
...
```

In the above the input/target columns represent the English and Romanian translation respectively.

All we need is three lines of code to train our model! By default, we use a mBART backbone for translation which requires a GPU to train.

```
# import our libraries
import flash
from flash import download_data
from flash.text import TranslationData, TranslationTask

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

# Organize the data
datamodule = TranslationData.from_files(
    train_file="data/wmt_en_ro/train.csv",
    valid_file="data/wmt_en_ro/valid.csv",
    test_file="data/wmt_en_ro/test.csv",
    input="input",
    target="target",
)

# 2. Build the task
model = TranslationTask()

# 4. Create trainer- in this case we need to run on gpus, `precision=16` boosts speed
```

(continues on next page)

(continued from previous page)

```
trainer = flash.Trainer(max_epochs=5, gpus=1, precision=16)

# 5. Finetune the task
trainer.finetune(model, datamodule=datamodule)

# 6. Save model to checkpoint
trainer.save_checkpoint("translation_model_en_ro.pt")
```

To run the example:

```
python flash_examples/finetuning/translation.py
```

11.4 Changing the backbone

You can change the model run by passing in the backbone parameter.

Note: When changing the backbone, make sure you pass in the same backbone to the Task and the Data object! Since this is a Seq2Seq task, make sure you use a Seq2Seq model.

```
datamodule = TranslationData.from_files(
    backbone="t5-small",
    train_file="data/wmt_en_ro/train.csv",
    valid_file="data/wmt_en_ro/valid.csv",
    test_file="data/wmt_en_ro/test.csv",
    input="input",
    target="target",
)

model = TranslationTask(backbone="t5-small")
```

11.5 API reference

11.5.1 TranslationTask

class flash.text.**TranslationTask**(*backbone='facebook/mbart-large-en-ro'*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=0.0003*, *val_target_max_length=128*, *num_beams=4*, *n_gram=4*, *smooth=False*)

Task for Sequence2Sequence Translation.

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*.
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **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*
- **n_gram** (bool) – Maximum n_grams to use in metric calculation. Defaults to *4*
- **smooth** (bool) – Apply smoothing in BLEU calculation. Defaults to *True*

property task

Override to define AutoConfig task specific parameters stored within the model.

Return type str

11.5.2 TranslationData

class flash.text.**TranslationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

Data module for Translation tasks.

classmethod TranslationData.**from_files**(*train_file*, *input='input'*, *target=None*, *filetype='csv'*, *backbone='facebook/mbart-large-en-ro'*, *val_file=None*, *test_file=None*, *predict_file=None*, *max_source_length=128*, *max_target_length=128*, *padding='max_length'*, *batch_size=8*, *num_workers=None*, *preprocess_cls=None*)

Creates a TranslateData object from files.

Parameters

- **train_file** – Path to training data.
- **input** (str) – The field storing the source translation text.
- **target** (Optional[str]) – The field storing the target translation text.
- **filetype** – .csv or .json
- **backbone** – Tokenizer backbone to use, can use any HuggingFace tokenizer.
- **val_file** – Path to validation data.
- **test_file** – Path to test data.
- **predict_file** – Path to predict data.
- **max_source_length** (int) – Maximum length of the source text. Any text longer will be truncated.
- **max_target_length** (int) – Maximum length of the target text. Any text longer will be truncated.
- **padding** (Union[str, bool]) – Padding strategy for batches. Default is pad to maximum length.

- **batch_size** (int) – The batchsize to use for parallel loading. Defaults to 8.
- **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 Darwin platform.

Returns The constructed data module.

Return type TranslateData

Examples:

```
train_df = pd.read_csv("train_data.csv")
tab_data = TabularData.from_df(train_df, target="fraud",
                               num_cols=["account_value"],
                               cat_cols=["account_type"])
```

CHAPTER

TWELVE

OBJECT DETECTION

12.1 The task

The object detection task identifies instances of objects of a certain class within an image.

12.2 Inference

The *ObjectDetector* is already pre-trained on COCO train2017, a dataset with 91 classes (123,287 images, 886,284 instances).

```
annotation{
    "id": int,
    "image_id": int,
    "category_id": int,
    "segmentation": RLE or [polygon],
    "area": float,
    "bbox": [x,y,width,height],
    "iscrowd": 0 or 1,
}

categories[{
    "id": int,
    "name": str,
    "supercategory": str,
}]
```

Use the *ObjectDetector* pretrained model for inference on any image tensor or image path using `predict()`:

```
from flash.vision import ObjectDetector

# 1. Load the model
detector = ObjectDetector()

# 2. Perform inference on an image file
predictions = detector.predict("path/to/image.png")
print(predictions)
```

Or on a random image tensor

```
# Perform inference on a random image tensor
import torch
images = torch.rand(32, 3, 1080, 1920)
predictions = detector.predict(images)
print(predictions)
```

For more advanced inference options, see *Predictions (inference)*.

12.3 Finetuning

To tailor the object detector to your dataset, you would need to have it in COCO Format, and then finetune the model.

Tip: You could also pass *trainable_backbone_layers* to `ObjectDetector` and train the model.

```
import flash
from flash.core.data import download_data
from flash.vision import ObjectDetectionData, ObjectDetector

# 1. Download the data
# 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/")

# 2. Load the Data
datamodule = ObjectDetectionData.from_coco(
    train_folder="data/coco128/images/train2017/",
    train_ann_file="data/coco128/annotations/instances_train2017.json",
    batch_size=2
)

# 3. Build the model
model = ObjectDetector(model="fasterrcnn", backbone="simclr-imagenet", num_
↪classes=datamodule.num_classes)

# 4. Create the trainer. Run thrice on data
trainer = flash.Trainer(max_epochs=3)

# 5. Finetune the model
trainer.finetune(model, datamodule)

# 6. Save it!
trainer.save_checkpoint("object_detection_model.pt")
```

12.4 Model

By default, we use the Faster R-CNN model with a ResNet-50 FPN backbone. We also support RetinaNet. The inputs could be images of different sizes. The model behaves differently for training and evaluation. For training, it expects both the input tensors as well as the targets. And during the evaluation, it expects only the input tensors and returns predictions for each image. The predictions are a list of boxes, labels, and scores.

12.5 Changing the backbone

By default, we use a ResNet-50 FPN backbone. You can change the backbone for the model by passing in a different backbone.

```
# 1. Organize the data
datamodule = ObjectDetectionData.from_coco(
    train_folder="data/coco128/images/train2017/",
    train_ann_file="data/coco128/annotations/instances_train2017.json",
    batch_size=2
)

# 2. Build the Task
model = ObjectDetector(model="retinanet", backbone="resnet101", num_
↪classes=datamodule.num_classes)
```

Available backbones:

- resnet18
- resnet34
- resnet50
- resnet101
- resnet152
- resnext50_32x4d
- resnext101_32x8d
- mobilenet_v2
- vgg11
- vgg13
- vgg16
- vgg19
- densenet121
- densenet169
- densenet161
- swav-imagenet
- simclr-imagenet

12.6 API reference

12.6.1 ObjectDetector

class flash.vision.**ObjectDetector**(*num_classes*, *model='fasterrcnn'*, *backbone=None*, *fpn=True*, *pretrained=True*, *pretrained_backbone=True*, *trainable_backbone_layers=3*, *anchor_generator=None*, *loss=None*, *metrics=None*, *optimizer=torch.optim.Adam*, *learning_rate=0.001*, ***kwargs*)

Object detection task

Ref: Lightning Bolts https://github.com/PyTorchLightning/lightning-bolts

Parameters

- **num_classes**¶ (int) – the number of classes for detection, including background
- **model**¶ (str) – 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**¶ (bool) – 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**¶ (bool) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers**¶ (int) – 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**¶ (Union[Callable, Module, Mapping, Sequence, None]) – The provided metrics. All metrics here will be logged to progress bar and the respective logger.
- **optimizer**¶ (Type[Optimizer]) – The optimizer to use for training. Can either be the actual class or the class name.
- **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

training_step(*batch*, *batch_idx*)

The training step. Overrides Task.training_step

Return type Any

12.6.2 ObjectDetectionData

class flash.vision.**ObjectDetectionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

classmethod ObjectDetectionData.**from_coco**(*train_folder=None*, *train_ann_file=None*, *train_transform=torchvision.transforms.ToTensor*, *val_folder=None*, *val_ann_file=None*, *val_transform=torchvision.transforms.ToTensor*, *test_folder=None*, *test_ann_file=None*, *test_transform=torchvision.transforms.ToTensor*, *batch_size=4*, *num_workers=None*, *preprocess_cls=None*, ***kwargs*)

CHAPTER

THIRTEEN

MODEL

class flash.core.model.**Task**(*model=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=5e-05*, *default_preprocess=None*, *default_postprocess=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** (Type[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*.
- **default_preprocess** (Optional[*Preprocess*]) – *Preprocess* to use as the default for this task.
- **default_postprocess** (Optional[*Postprocess*]) – *Postprocess* to use as the default for this task.

build_data_pipeline(*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*.

property data_pipeline

The current *DataPipeline*. If set, the new value will override the *Task* defaults. See *build_data_pipeline()* for more details on the resolution order.

Return type *DataPipeline*

predict(*x*, *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*)

The training/validation/test step. Override for custom behavior.

Return type Any

CHAPTER

FOURTEEN

DATA

14.1 Terminology

Here are common terms you need to be familiar with:

Table 1: Terminology

Term	Definition
`DataModule`	The `DataModule` contains the dataset, transforms and dataloaders.
`DataPipeline`	The `DataPipeline` is Flash internal object to manage `Preprocess` and `Postprocess` objects.
`Preprocess`	**The `Preprocess` provides a simple hook-based API to encapsulate your pre-processing logic.** The `Preprocess` provides multiple hooks such as `load_data()` and `load_sample()` which are used to replace a traditional *Dataset* logic. Flash DataPipeline contains a system to call the right hooks when needed. The `Preprocess` hooks covers from data-loading to model forwarding.
`Postprocess`	**The `Postprocess` provides a simple hook-based API to encapsulate your post-processing logic.** The `Postprocess` hooks covers from model outputs to predictions export.

14.2 How to use out-of-the-box flashdatamodules

Flash provides several DataModules with helpers functions. Checkout the *Image Classification* section or any other tasks to learn more about them.

14.3 Data Processing

Currently, it is common practice to implement a Dataset and provide them to a DataLoader.

However, after model training, it requires a lot of engineering overhead to make inference on raw data and deploy the model in production environnement. Usually, extra processing logic should be added to bridge the gap between training data and raw data.

The `Preprocess` and `Postprocess` classes can be used to store the data as well as the preprocessing and post-processing transforms.

By providing a series of hooks that can be overridden with custom data processing logic, the user has 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.

Note: `[WIP]` We are currently working on a new feature to make *Preprocess*

and *Postprocess* automatically deployable from checkpoints as

`Endpoints` or `BatchTransformJob`. Stay tuned !

14.4 How to customize existing datamodules

Flash DataModule can receive directly dataset as follow:

Example:

```
from flash.data.data_module import DataModule

dm = DataModule(train_dataset=MyDataset(train=True))
trainer = Trainer(fast_dev_run=True)
trainer.fit(model, data_module=dm)
```

In order to customize Flash to your need, you need to know what are *DataModule* and *Preprocess* responsibilities.

Note: At this point, we strongly encourage the readers to quickly check the *Preprocess* API before getting further.

The *DataModule* provides `classmethod` helpers to build *Preprocess* and *DataPipeline*, generate Flash Internal `AutoDataset` and populate DataLoaders with them.

The *Preprocess* contains the processing logic related to a given task. Users can easily override hooks to customize a built-in *Preprocess* for their needs.

Example:

```
from flash.vision import ImageClassificationData, ImageClassifier,␣
↪ImageClassificationPreprocess

class CustomImageClassificationPreprocess(ImageClassificationPreprocess):

    # Assuming you have images in numpy format,
    # just override ``load_sample`` hook and add your own logic.
    @staticmethod
    def load_sample(sample) -> Tuple[Image.Image, int]:
```

(continues on next page)

(continued from previous page)

```
        # By default, ``ImageClassificationPreprocess`` expects
        # ``.png`` or ``.jpg`` to be loaded into PIL Image.
        numpy_image_path, label = sample
        return np.load(numpy_image_path), sample

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

14.5 Custom Preprocess + Datamodule

The example below shows a very simple `ImageClassificationPreprocess` with a `ImageClassificationDataModule`.

14.5.1 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:

```
preprocess = ...

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

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

trainer.fit(model, dm)
```

14.5.2 2. The DataModule

Secondly, let's implement the `ImageClassificationDataModule` from_folders classmethod.

Example:

```
from flash.data.data_module import DataModule

class ImageClassificationDataModule(DataModule):

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

    @classmethod
    def from_folders(
        cls,
        train_folder: Optional[str],
        val_folder: Optional[str],
        test_folder: Optional[str],
        predict_folder: Optional[str],
        preprocess: Optional[Preprocess] = None,
        **kwargs
    ):

        preprocess = preprocess or cls.preprocess_cls()

        # {stage}_load_data_input will be given to your
        # ``Preprocess`` ``{stage}_load_data`` function.
        return cls.from_load_data_inputs(
                train_load_data_input=train_folder,
                val_load_data_input=val_folder,
                test_load_data_input=test_folder,
                predict_load_data_input=predict_folder,
                preprocess=preprocess,  # DON'T FORGET TO PASS THE CREATED PREPROCESS
                **kwargs,
        )
```

14.5.3 3. The Preprocess

Finally, implement your custom `ImageClassificationPreprocess`.

Example:

```
import os
import numpy as np
from flash.data.process import Preprocess
from PIL import Image
import torchvision.transforms as T
from torch import Tensor
from torchvision.datasets.folder import make_dataset

# Subclass ``Preprocess``
class ImageClassificationPreprocess(Preprocess):

    to_tensor = T.ToTensor()

    def load_data(self, folder: str, dataset: AutoDataset) -> Iterable:
```

(continues on next page)

(continued from previous page)

```
        # The AutoDataset is optional but can be useful to save some metadata.

        # metadata contains the image path and its corresponding label with the␣
↪following structure:
        # [(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 metadata

    def predict_load_data(self, predict_folder: str) -> Iterable:
        # This returns [image_path_1, ... image_path_m].
        return os.listdir(folder)

    def load_sample(self, sample: Union[str, Tuple[str, int]]) -> Tuple[Image, int]
        if self.predicting:
            return Image.open(image_path)
        else:
            image_path, label = sample
            return Image.open(image_path), label

    def to_tensor_transform(
        self,
        sample: Union[Image, Tuple[Image, int]]
    ) -> Union[Tensor, Tuple[Tensor, int]]:

        if self.predicting:
            return self.to_tensor(sample)
        else:
            return self.to_tensor(sample[0]), sample[1]
```

Note: Currently, Flash Tasks are implemented using *Preprocess* and *Postprocess*. However, it is not a hard requirement and one can still use `Dataset`, but we highly recommend using *Preprocess* and *Postprocess* instead.

14.6 API reference

14.6.1 Preprocess

class flash.data.process.**Preprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*)

The *Preprocess* encapsulates all the data processing and loading logic that should run before the data is passed to the model.

It is particularly relevant when you want to provide an end to end implementation which works with 4 different stages: `train`, `validation`, `test`, and inference (`predict`).

You can override any of the preprocessing hooks to provide custom functionality. All hooks default to no-op (except the collate which is PyTorch default collate)

The *Preprocess* supports the following hooks:

- **`load_data`: Function to receiving some metadata to generate a Mapping from.** Example:

```
* Input: Receive a folder path:

* Action: Walk the folder path to find image paths and their associated
↪labels.

* Output: Return a list of image paths and their associated labels.
```

- **`load_sample`: Function to load a sample from metadata sample.** Example:

```
* Input: Receive an image path and its label.

* Action: Load a PIL Image from received image_path.

* Output: Return the PIL Image and its label.
```

- **`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.** Example:

```
* Input: Receive a list of augmented tensored images and their respective␣
↪labels.

* 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 substracting the mean
    and dividing by the standard deviation from ImageNet.

* Output: Return a normalized augmented batch of images and their labels.
```

Note: By default, each hook will be no-op execpt the collate which is PyTorch default collate. To customize them, just override the hooks and `Flash` will take care of calling them at the right moment.

Note: The `per_sample_transform_on_device` and `per_batch_transform` are mutually exclusive as it will impact performances.

To change the processing behavior only on specific stages, you can prefix all the above hooks adding `train`, `val`, `test` or `predict`.

For example, is useful to encapsulate `predict` logic as labels aren't availabled at inference time.

Example:

```
class CustomPreprocess(Preprocess):

    def predict_load_data(cls, data: Any, dataset: Optional[Any] = None) ->␣
↪Mapping:
        # logic for predict data only.
```

Each hook is aware of the Trainer `running stage` through booleans as follow.

This is useful to adapt a hook internals for a stage without duplicating code.

Example:

```
class CustomPreprocess(Preprocess):

    def load_data(cls, data: Any, dataset: Optional[Any] = None) -> Mapping:

        if self.training:
            # logic for train

        elif self.validating:
            # logic from validation

        elif self.testing:
            # logic for test
```

(continues on next page)

(continued from previous page)

```
        elif self.predicting:
            # logic for predict
```

Note: It is possible to wrap a `Dataset` within a *`load_data()`* function. However, we don't recommend to do as such as it is better to rely entirely on the hooks.

Example:

```
from torchvision import datasets

class CustomPreprocess(Preprocess):

    def load_data(cls, path_to_data: str) -> Iterable:

        return datasets.MNIST(path_to_data, download=True, transform=transforms.
↪ToTensor())
```

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

Loads entire data from Dataset. The input `data` can be anything, but you need to return a Mapping.

Example:

```
# data: "."
# output: [("./cat/1.png", 1), ..., ("./dog/10.png", 0)]

output: Mapping = load_data(data)
```

Return type `Mapping`

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

Loads single sample from dataset

Return type `Any`

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

14.6.2 Postprocess

class flash.data.process.**Postprocess**(*save_path=None*)

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

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

save_data(*data*, *path*)
Saves all data together to a single path.

Return type None

save_sample(*sample*, *path*)
Saves each sample individually to a given path.

Return type None

uncollate(*batch*)
Uncollates a batch into single samples. Tries to preserve the type whereever possible.

Return type Any

14.6.3 DataPipeline

class flash.data.data_pipeline.**DataPipeline**(*preprocess=None*, *postprocess=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
```

14.6.4 DataModule

class flash.data.data_module.**DataModule**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *batch_size=1*, *num_workers=0*)

Basic DataModule class for all Flash tasks

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 to predict model performance. Defaults to None.
- **num_workers** (Optional[int]) – The number of workers to use for parallelized loading. Defaults to None.
- **predict_ds** – Dataset for predicting. Defaults to None.
- **batch_size** (int) – The batch size to be used by the DataLoader. Defaults to 1.
- **num_workers** – The number of workers to use for parallelized loading. Defaults to None which equals the number of available CPU threads, or 0 for Darwin platform.

static configure_data_fetcher(**args*, ***kwargs*)

This function is used to configure a *BaseDataFetcher*. Override with your custom one.

Return type *BaseDataFetcher*

classmethod from_load_data_inputs(*train_load_data_input=None*, *val_load_data_input=None*, *test_load_data_input=None*, *predict_load_data_input=None*, *preprocess=None*, *postprocess=None*, ***kwargs*)

This functions is an helper to generate a `DataModule` from a `DataPipeline`.

Parameters

- **cls**¶ – `DataModule` subclass
- **train_load_data_input**¶ (`Optional`[`Any`]) – Data to be received by the `train_load_data` function from this *Preprocess*
- **val_load_data_input**¶ (`Optional`[`Any`]) – Data to be received by the `val_load_data` function from this *Preprocess*
- **test_load_data_input**¶ (`Optional`[`Any`]) – Data to be received by the `test_load_data` function from this *Preprocess*
- **predict_load_data_input**¶ (`Optional`[`Any`]) – Data to be received by the `predict_load_data` function from this *Preprocess*
- **kwargs**¶ – Any extra arguments to instantiate the provided `DataModule`

Return type *DataModule*

property predict_dataset

This property returns the predict dataset

Return type `Optional`[`Dataset`]

show_predict_batch(*reset=True*)

This function is used to visualize a batch from the predict dataloader.

Return type `None`

show_test_batch(*reset=True*)

This function is used to visualize a batch from the test dataloader.

Return type `None`

show_train_batch(*reset=True*)

This function is used to visualize a batch from the train dataloader.

Return type `None`

show_val_batch(*reset=True*)

This function is used to visualize a batch from the validation dataloader.

Return type `None`

property test_dataset

This property returns the test dataset

Return type `Optional`[`Dataset`]

property train_dataset

This property returns the train dataset

Return type `Optional`[`Dataset`]

property val_dataset

This property returns the validation dataset

Return type Optional[Dataset]

14.7 How it works behind the scenes

14.7.1 Preprocess

Note: The `load_data` and `load_sample` will be used to generate an AutoDataset object.

Here is the `AutoDataset` pseudo-code.

Example:

```
from pytorch_lightning.trainer.states import RunningStage

class AutoDataset
    def __init__(
        self,
        data: Any,
        load_data: Optional[Callable] = None,
        load_sample: Optional[Callable] = None,
        data_pipeline: Optional['DataPipeline'] = None,
        running_stage: Optional[RunningStage] = None
    ) -> None:

        self.preprocess = data_pipeline._preprocess_pipeline
        self.preprocessed_data: Iterable = self.preprocess.load_data(data)

    def __getitem__(self, index):
        return self.preprocess.load_sample(self.preprocessed_data[index])

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

Note: The `pre_tensor_transform`, `to_tensor_transform`, `post_tensor_transform`, `collate`, `per_batch_transform` are injected as the `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.data.batch._PreProcessor`.
def collate_fn(samples: Sequence[Any]) -> Any:

    # This will be wrapped into a :class:`~flash.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.data.process.Preprocess.per_sample_transform_on_device` hook is␣
↪overridden,
```

(continues on next page)

(continued from previous page)

```
    # 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.data.batch._PreProcessor`
def collate_fn(samples: Sequence[Any]) -> Any:

    # 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)
```

14.7.2 Postprocess

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

First, the `per_batch_transform` hooks will be applied on the batch predictions. Then the `uncollate` will split the batch into individual predictions. Finally, the `per_sample_transform` will be applied on each prediction.

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.data.batch._PreProcessor`
def uncollate_fn(batch: Any) -> Any:

    batch = per_batch_transform(batch)

    samples = uncollate(batch)
```

(continues on next page)

(continued from previous page)

```
    return [per_sample_transform(sample) for sample in samples]

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

CHAPTER

FIFTEEN

CALLBACK

15.1 Flash Callback

FlashCallback is an extension of the PyTorch Lightning 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.

Example:

```
trainer = Trainer(callbacks=[MyCustomCallback()])
```

15.2 Available Callbacks

15.2.1 BaseDataFetcher

class flash.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.data.callback import BaseDataFetcher
from flash.data.data_module import DataModule


class PrintData(BaseDataFetcher):

    def print(self):
        print(self.batches)

class CustomDataModule(DataModule):

    @staticmethod
    def configure_data_fetcher():
        return PrintData()
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def from_inputs(
        cls,
        train_data: Any,
        val_data: Any,
        test_data: Any,
        predict_data: Any) -> "CustomDataModule":

        preprocess = cls.preprocess_cls()

        return cls.from_load_data_inputs(
            train_load_data_input=train_data,
            val_load_data_input=val_data,
            test_load_data_input=test_data,
            predict_load_data_input=predict_data,
            preprocess=preprocess,
            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': {},
    'test': {},
    'val': {},
    'predict': {}
}
```

enable()
This function is used to enable to BaseDataFetcher

15.2.2 BaseVisualization

class flash.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.vision import ImageClassificationData
from flash.data.base_viz import BaseVisualization

class CustomBaseVisualization(BaseVisualization):

    def show_load_sample(self, samples: List[Any], running_stage):
        # plot samples

    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
        {
            '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*)
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.

15.3 API reference

15.3.1 FlashCallback

class flash.data.callback.**FlashCallback**(**args*, ***kwargs*)

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

CHAPTER

SIXTEEN

REGISTRY

16.1 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.

16.1.1 1. Imports

Example:

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

16.1.2 2. Init a Registry

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

Example:

```
from flash.vision import ImageClassifier
from flash.core.registry import FlashRegistry

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

    backbones = FlashRegistry("backbones")
```

16.1.3 3. Adding new functions

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

Example:

```
# Option 1: Used with partial.
def fn(backbone: str):
    # Create backbone and backbone output dimension (`num_features`)
    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/my_
↪backbone")


# Option 2: Using decorator.
@MyImageClassifier.backbones(name="username/my_backbone")
def fn():
    # Create backbone and backbone output dimension (`num_features`)
    return backbone, num_features
```

16.1.4 4. Accessing registered functions

You can now access your function from your task!

Example:

```
# 3.b Optional: List available backbones
print(MyImageClassifier.available_backbones())
# out: ["username/my_backbone"]

# 4. Build the model
model = MyImageClassifier(backbone="username/my_backbone", num_classes=2)
```

16.1.5 5. Pre-registered ones

Flash provides already populated registries containing lot of available backbones.

Example:

```
from flash.vision.backbones import IMAGE_CLASSIFIER_BACKBONES, OBJ_DETECTION_BACKBONES

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

16.2 Flash Registry

16.2.1 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]]

CHAPTER

SEVENTEEN

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. If you want to train the task from scratch instead, pass *pretrained=False* parameter when creating your task. Then, use the *fit()* method to train your model.

```
import flash
from flash import download_data
from flash.vision import ImageClassificationData, ImageClassifier

# 1. download and organize the data
download_data("https://download.pytorch.org/tutorial/hymenoptera_data.zip", 'data/')

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

# 2. build the task, and turn off pre-training
task = ImageClassifier(num_classes=2, pretrained=False)

# 3. train!
trainer = flash.Trainer()
trainer.fit(model, data)
```

17.1 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.

17.1.1 Trainer API

class flash.core.trainer.**Trainer**(**args*, ***kwargs*)

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 finetuning callback subclassing pytorch_lightning.callbacks.BaseFinetuning.

  Currently, 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

CHAPTER

EIGHTEEN

FINETUNING

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

18.1 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

18.2 3 steps to finetune in Flash

All Flash tasks have a pre-trained backbone that was already trained on large datasets such as ImageNet. Finetuning on already pretrained models decrease training time significantly.

To finetune using Flash, follow these 3 steps:

1. Load your data and organize it using a DataModule customized for the task.
2. Pick a Task which has all the state-of-the-art built in (example: *`ImageClassifier`*).
3. Choose a Finetune strategy and call the *`finetune()`* method

Here are the steps in code

```
import flash
from flash import download_data
from flash.vision import ImageClassificationData, ImageClassifier

# 1. download and organize the data
download_data("https://download.pytorch.org/tutorial/hymenoptera_data.zip", 'data/')

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

# 2. build the model
model = ImageClassifier(num_classes=2)

# 3. Build the trainer and finetune! In this case, using the no_freeze strategy
trainer = flash.Trainer()
trainer.finetune(task, data, strategy="no_freeze")
```

Tip: If you have a large dataset and prefer to train from scratch, see the *Training from scratch* guide.

18.3 Using a finetuned model

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

```
predictions = task.predict('data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg')
print(predictions)
```

Or use a different checkpoint for prediction

```
# Save the checkpoint while training.
trainer.save_checkpoint("image_classification_model.pt")

# load the finetuned model
classifier = ImageClassifier.load_from_checkpoint('image_classification_model.pt')

# predict!
predictions = classifier.predict('data/hymenoptera_data/val/bees/65038344_52a45d090d.
↪jpg')
print(predictions)
```

18.4 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.

18.4.1 no_freeze

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

```
trainer.finetune(task, data, strategy='no_freeze')
```

In pseudocode, this looks like:

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

backbone.unfreeze()
head.unfreeze()

train(backbone, head)
```

18.4.2 freeze

The freeze strategy keeps the backbone frozen throughout.

```
trainer.finetune(task, data, strategy='freeze')
```

The psedocode looks like:

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

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

train(backbone, head)
```

18.4.3 freeze_unfreeze

In this strategy, the backbone is frozen for 10 epochs then unfrozen.

```
trainer.finetune(model, data, strategy='freeze_unfreeze')
```

```
from flash.core.finetuning import FreezeUnfreeze

# finetune for 10 epochs. Backbone will be frozen for 5 epochs.
trainer = flash.Trainer(max_epochs=10)
trainer.finetune(model, data, strategy=FreezeUnfreeze(unfreeze_epoch=5))
```

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)
```

18.5 Advanced strategies

Every finetune strategy can also be customized.

18.5.1 freeze_unfreeze

In this strategy, the backbone is frozen for x epochs then unfrozen.

Here we unfreeze the backbone at epoch 11.

```
from flash.core.finetuning import FreezeUnfreeze

trainer = flash.Trainer(max_epochs=10)
trainer.finetune(model, data, strategy=FreezeUnfreeze(unfreeze_epoch=11))
```

18.5.2 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

# finetune for 10 epochs.
trainer = flash.Trainer(max_epochs=10)
trainer.finetune(model, data, 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()
```

18.6 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_at_epoch: int = 5, train_bn: bool = True):
        # this will set self.attr_names as ["feature_extractor"]
        super().__init__("feature_extractor", train_bn)
        self._unfreeze_at_epoch = unfreeze_at_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, feature_extractor will start to be trained.
        if current_epoch == self._unfreeze_at_epoch:
            self.unfreeze_and_add_param_group(
                module=pl_module.feature_extractor,
                optimizer=optimizer,
                train_bn=True,
            )

# Init the trainer
trainer = flash.Trainer(max_epochs=10)

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

CHAPTER

NINETEEN

PREDICTIONS (INFERENCE)

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

19.1 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 import Trainer
from flash.core.data import download_data
from flash.vision import ImageClassificationData, 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)
```

19.2 Predict on a csv file

```
from flash.core.data 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)
```

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

P

S

T

U

V