Flash

PyTorch Lightning

May 20, 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 advanced features PyTorch Lightning offers.

1.1 Why Flash?

1.1.1 For getting started with Deep Learning

Easy to learn

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

Easy to scale

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

Easy to upskill

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

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

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

1.1.2 For Deep learning research

Quickest way to a baseline

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

Flexibility where you want it

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

Standard best practices

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

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

1.2 Tasks

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

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

Here are examples of tasks:

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

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

1.3 Inference

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

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

Here's an example of inference:

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

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

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

We get the following output:

```
[1, 1, 0]
```

1.4 Finetuning

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

To use a Task for finetuning:

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

Here's an example of finetuning.

```
from pytorch_lightning import seed_everything

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

# set the random seeds.
seed_everything(42)

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

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

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

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

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

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

1.4.1 Using a finetuned model

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

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

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

We get the following output:

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

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

```
from flash.image import ImageClassifier

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

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

1.5 Training

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

To train a task from scratch:

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

Here's an example:

```
from pytorch_lightning import seed_everything

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

# set the random seeds.
seed_everything(42)

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

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

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

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

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

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

1.6 A few Built-in Tasks

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

More tasks coming soon!

1.6.1 Contribute a task

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

Join our Slack to get help becoming a contributor!

CHAPTER

TWO

INSTALLATION

Flash is tested on Python 3.6+, and PyTorch 1.6

2.1 Install with pip/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*.

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

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

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

3.1 1. Imports

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

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

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

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

# set the random seeds.
seed_everything(42)

ND = np.ndarray
```

3.2 2. The Task: Linear regression

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

```
class RegressionTask(flash.Task):

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        return self.model(x)
```

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

3.2.1 Where is the training step?

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

Here is the pseudo code behind *Task* step:

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

3.3 3.a The DataSource API

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

```
class NumpyDataSource(DataSource[Tuple[ND, ND]]):

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

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

3.4 3.b The Preprocess API

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

Note: Why introduce *Preprocess* ?

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

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

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

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

```
class NumpyPreprocess(Preprocess):

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

3.5 3.c The DataModule API

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

```
class NumpyDataModule(flash.DataModule):

    preprocess_cls = NumpyPreprocess
```

You now have a new customized Flash Task! Congratulations !

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

3.6 4. Fitting

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

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

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

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

trainer = flash.Trainer(max_epochs=20, progress_bar_refresh_rate=20, checkpoint_
↪callback=False)
trainer.fit(model, datamodule=datamodule)
```

3.7 5. Predicting

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

```
predict_data = np.array([
    [ 0.0199,  0.0507,  0.1048,  0.0701, -0.0360, -0.0267, -0.0250, -0.0026,  0.0037,␣
↪ 0.0403],
    [-0.0128, -0.0446,  0.0606,  0.0529,  0.0480,  0.0294, -0.0176,  0.0343,  0.0702,␣
↪ 0.0072],
    [ 0.0381,  0.0507,  0.0089,  0.0425, -0.0428, -0.0210, -0.0397, -0.0026, -0.0181,␣
↪ 0.0072],
    [-0.0128, -0.0446, -0.0235, -0.0401, -0.0167,  0.0046, -0.0176, -0.0026, -0.0385,␣
↪-0.0384],
    [-0.0237, -0.0446,  0.0455,  0.0907, -0.0181, -0.0354,  0.0707, -0.0395, -0.0345,␣
↪-0.0094],
])

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

We get the following output:

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

CHAPTER

FOUR

FROM FLASH TO LIGHTNING

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

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

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

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

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

Table 1: Abstraction levels

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

4.1 Using an out-of-the-box task

Tasks can come from a variety of places:

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

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

- data science
- analysis
- applied research

4.2 Using the Generic Task

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

This is great for:

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

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

4.3 Building a custom task

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

This is great for:

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

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

4.4 Building a LightningModule

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

This is great for:

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

CHAPTER

FIVE

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

```
import os

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

from flash.core.classification import ClassificationTask
from flash.core.data.utils import download_data

_PATH_ROOT = os.path.dirname(os.path.dirname(__file__))

# 1. Download the data
download_data("https://www.di.ens.fr/~lelarge/MNIST.tar.gz", os.path.join(_PATH_ROOT,
↪'data'))

# 2. Load a basic backbone
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ReLU(),
    nn.Linear(128, 10),
)

# 3. Load a dataset
dataset = datasets.MNIST(
    os.path.join(_PATH_ROOT, 'data'),
    download=False,
    transform=transforms.ToTensor(),
)

# 4. Split the data randomly
train, val, test = random_split(dataset, [50000, 5000, 5000])  # type: ignore
```

(continues on next page)

(continued from previous page)

```
# 5. Create the model
classifier = ClassificationTask(
    model,
    loss_fn=nn.functional.cross_entropy,
    optimizer=optim.Adam,
    learning_rate=10e-3,
)

# 6. Create the trainer
trainer = pl.Trainer(
    max_epochs=10,
    limit_train_batches=128,
    limit_val_batches=128,
)

# 7. Train the model
trainer.fit(classifier, DataLoader(train), DataLoader(val))

# 8. Test the model
results = trainer.test(classifier, test_dataloaders=DataLoader(test))
```

5.2 API reference

5.2.1 Task

class flash.core.model.**Task**(*model=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=5e-05*, *preprocess=None*, *postprocess=None*, *serializer=None*)

A general Task.

Parameters

- **model** (Optional[Module]) – Model to use for the task.
- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training, defaults to torch.optim.Adam.
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **learning_rate** (float) – Learning rate to use for training, defaults to 5e-5.
- **preprocess** (Optional[*Preprocess*]) – *Preprocess* to use as the default for this task.
- **postprocess** (Optional[*Postprocess*]) – *Postprocess* to use as the default for this task.

build_data_pipeline(*data_source=None*, *data_pipeline=None*)

Build a *DataPipeline* incorporating available *Preprocess* and *Postprocess* objects. These will be overridden in the following resolution order (lowest priority first):

- Lightning `Datamodule`, either attached to the *Trainer* or to the *Task*.
- *Task* defaults given to `Task.__init__()`.
- *Task* manual overrides by setting *data_pipeline*.
- *DataPipeline* passed to this method.

Parameters **data_pipeline** (`Optional`[*DataPipeline*]) – Optional highest priority source of *Preprocess* and *Postprocess*.

Return type `Optional`[*DataPipeline*]

Returns The fully resolved *DataPipeline*.

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*

get_num_training_steps()

Total training steps inferred from datamodule and devices.

Return type `int`

predict(*x*, *data_source=None*, *data_pipeline=None*)

Predict function for raw data or processed data

Parameters

- **x** (`Any`) – Input to predict. Can be raw data or processed data. If str, assumed to be a folder of data.
- **data_pipeline** (`Optional`[*DataPipeline*]) – Use this to override the current data pipeline

Return type `Any`

Returns The post-processed model predictions

property serializer

The current *Serializer* associated with this model. If this property was set to a mapping (e.g. `.serializer = {'output1': SerializerOne()}`) then this will be a `MappingSerializer`.

Return type `Optional`[*Serializer*]

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 describes 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()`:

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

model.serializer = Probabilities()
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/")
```

(continues on next page)

(continued from previous page)

```
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 to train our task!

```
import torchvision
from torch import nn

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

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

(continues on next page)

(continued from previous page)

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


# 3.a Optional: Register a custom backbone
# This is useful to create new backbone and make them accessible from␣
↪`ImageClassifier`
@ImageClassifier.backbones(name="resnet18")
def fn_resnet(pretrained: bool = True):
    model = torchvision.models.resnet18(pretrained)
    # remove the last two layers & turn it into a Sequential model
    backbone = nn.Sequential(*list(model.children())[:-2])
    num_features = model.fc.in_features
    # backbones need to return the num_features to build the head
    return backbone, num_features


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

# 4. Build the model
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes,␣
↪serializer=Labels())

# 5. Create the trainer
trainer = flash.Trainer(max_epochs=1, limit_train_batches=1, limit_val_batches=1)

# 6. Train the model
trainer.finetune(model, datamodule=datamodule, strategy=FreezeUnfreeze(unfreeze_
↪epoch=1))

# 7a. Predict what's on a few images! ants or bees?

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

predictions = model.predict([
    "data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg",
    "data/hymenoptera_data/val/bees/590318879_68cf112861.jpg",
    "data/hymenoptera_data/val/ants/540543309_ddbb193ee5.jpg",
])
print(predictions)

datamodule = ImageClassificationData.from_folders(predict_folder="data/hymenoptera_
↪data/predict/")

# 7b. Or generate predictions with a whole folder!
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)

# 8. 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.

```
# 1. organize the data
data = ImageClassificationData.from_folders(
    train_folder="data/hymenoptera_data/train/",
    val_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
- TIMM (130+ PyTorch Image Models)

6.5 API reference

6.5.1 ImageClassifier

class flash.image.**ImageClassifier**(*num_classes*, *backbone='resnet18'*, *backbone_kwargs=None*, *head=None*, *pretrained=True*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.001*, *multi_label=False*, *serializer=None*)

Task that classifies images.

Use a built in backbone

Example:

```
from flash.image import ImageClassifier

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

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

Example:

```
from flash.image 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** (Optional[Callable]) – Loss function for training, defaults to torch.nn.functional.cross_entropy().
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training, defaults to torch.optim.SGD.
- **metrics** (Union[Metric, 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.
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

6.5.2 ImageClassificationData

class flash.image.**ImageClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

Data module for image classification tasks.

class flash.image.**ImageClassificationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(196, 196)*)

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()*:

```
import torch

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

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

# 2. Create an ImageEmbedder with swav trained on imagenet.
# Check out SWAV: https://lightning-bolts.readthedocs.io/en/latest/self_supervised_
↪models.html#swav
embedder = ImageEmbedder(backbone="swav-imagenet", embedding_dim=128)

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

# 4. Print embeddings shape
print(embeddings[0].shape)

# 5. Create a tensor random image
random_image = torch.randn(1, 3, 244, 244)

# 6. Generate an embedding from this random image.
embeddings = embedder.predict(random_image, data_source="tensors")

# 7. Print embeddings shape
print(embeddings[0].shape)
```

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

7.3 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/",
    val_folder="data/hymenoptera_data/val/",
)

# 2. build the task
embedder = ImageEmbedder(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
- TIMM (130+ PyTorch Image Models)

7.4 API reference

7.4.1 ImageEmbedder

class flash.image.**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().

CHAPTER

EIGHT

MULTI-LABEL IMAGE CLASSIFICATION

8.1 The task

Multi-label classification is the task of assigning a number of labels from a fixed set to each data point, which can be in any modality. In this example, we will look at the task of trying to predict the movie genres from an image of the movie poster.

8.2 The data

The data we will use in this example is a subset of the awesome movie poster genre prediction data set from the paper "Movie Genre Classification based on Poster Images with Deep Neural Networks" by Wei-Ta Chu and Hung-Jui Guo, resized to 128 by 128. Take a look at their paper (and please consider citing their paper if you use the data) here: www.cs.ccu.edu.tw/~wtchu/projects/MoviePoster/.

8.3 Inference

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

We can use the *ImageClassifier* model (pretrained on our data) for inference on any string sequence using `predict()`. We can also add a simple visualisation by extending *BaseVisualization*, like this:

```
import os
from typing import Any

import torchvision.transforms.functional as T
from torchvision.utils import make_grid

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

# 1. Download the data
# This is a subset of the movie poster genre prediction data set from the paper
# "Movie Genre Classification based on Poster Images with Deep Neural Networks" by␣
↪Wei-Ta Chu and Hung-Jui Guo.
```

(continues on next page)

(continued from previous page)

```
# Please consider citing their paper if you use it. More here: https://www.cs.ccu.edu.
↪tw/~wtchu/projects/MoviePoster/
download_data("https://pl-flash-data.s3.amazonaws.com/movie_posters.zip", "data/")


# 2. Define our custom visualisation and datamodule
class CustomViz(BaseVisualization):

    def show_per_batch_transform(self, batch: Any, _) -> None:
        images = batch[0]["input"]
        image = make_grid(images, nrow=2)
        image = T.to_pil_image(image, 'RGB')
        image.show()


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

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

# 4b. Or generate predictions with a whole folder!
datamodule = ImageClassificationData.from_folders(
    predict_folder="data/movie_posters/predict/",
    data_fetcher=CustomViz(),
    image_size=(128, 128),
)

predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)

# 5. Show some data (unless we're just testing)!
datamodule.show_predict_batch("per_batch_transform")
```

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

8.4 Finetuning

Now let's look at how we can finetune a model on the movie poster data. 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 images and a `metadata.csv` which stores the labels.

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

The `metadata.csv` files in each folder contain our labels, so we need to create a function (`load_data`) to extract the list of images and associated labels:

```
import os.path as osp
from typing import List, Tuple

import pandas as pd
from torchmetrics import F1

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

# 1. Download the data
# This is a subset of the movie poster genre prediction data set from the paper
# "Movie Genre Classification based on Poster Images with Deep Neural Networks" by␣
↪Wei-Ta Chu and Hung-Jui Guo.
# Please consider citing their paper if you use it. More here: https://www.cs.ccu.edu.
↪tw/~wtchu/projects/MoviePoster/
download_data("https://pl-flash-data.s3.amazonaws.com/movie_posters.zip", "data/")

# 2. Load the data
genres = ["Action", "Romance", "Crime", "Thriller", "Adventure"]


def load_data(data: str, root: str = 'data/movie_posters') -> Tuple[List[str],␣
↪List[List[int]]]:
    metadata = pd.read_csv(osp.join(root, data, "metadata.csv"))
    return ([osp.join(root, data, row['Id'] + ".jpg") for _, row in metadata.
↪iterrows()],
            [[int(row[genre]) for genre in genres] for _, row in metadata.iterrows()])
```

(continues on next page)

(continued from previous page)

```
train_files, train_targets = load_data('train')
test_files, test_targets = load_data('test')

datamodule = ImageClassificationData.from_files(
    train_files=train_files,
    train_targets=train_targets,
    test_files=test_files,
    test_targets=test_targets,
    val_split=0.1,  # Use 10 % of the train dataset to generate validation one.
    image_size=(128, 128),
)

# 3. Build the model
model = ImageClassifier(
    backbone="resnet18",
    num_classes=len(genres),
    multi_label=True,
    metrics=F1(num_classes=len(genres)),
)

# 4. Create the trainer. Train on 2 gpus for 10 epochs.
trainer = flash.Trainer(max_epochs=10)

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

# 6. Predict what's on a few images!
# Serialize predictions as labels, low threshold to see more predictions.
model.serializer = Labels(genres, multi_label=True, threshold=0.25)

predictions = model.predict([
    "data/movie_posters/predict/tt0085318.jpg",
    "data/movie_posters/predict/tt0089461.jpg",
    "data/movie_posters/predict/tt0097179.jpg",
])

print(predictions)

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

For more backbone options, see *Image Classification*.

CHAPTER

NINE

SUMMARIZATION

9.1 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.

9.2 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:

```
from pytorch_lightning import Trainer

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

# 1. Download the 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")

# 2a. Summarize an article!
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
```

(continues on next page)

(continued from previous page)

```
    for 20 years, said Camilla had been ""nice and pleasant"" when she purchased the␣
↪fruit.
    ""She asked me what was nice, what would I recommend, and I said we've got some␣
↪nice mangoes.
    She asked me were they ripe and I said yes - they're from the Dominican Republic."
↪"
    Mr Chong is one of 170 local retailers who accept the Brixton Pound.
    Customers exchange traditional pound coins for Brixton Pounds and then spend them␣
↪at the market
    or in participating shops.
    During the visit, Prince Charles spent time talking to youth worker Marcus West,␣
↪who works with children
    nearby on an estate off Coldharbour Lane. Mr West said:
    ""He's on the level, really down-to-earth. They were very cheery. The prince is a␣
↪lovely man.""
    He added: ""I told him I was working with young kids and he said, 'Keep up all␣
↪the good work.'""
    Prince Charles also visited the Railway Hotel, at the invitation of his charity␣
↪The Prince's Regeneration Trust.
    The trust hopes to restore and refurbish the building,
    where once Jimi Hendrix and The Clash played, as a new community and business␣
↪centre."
    """
])
print(predictions)

# 2b. Or generate summaries from a sheet file!
datamodule = SummarizationData.from_csv(
    "input",
    predict_file="data/xsum/predict.csv",
)
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

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

9.3 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.

```
import torch

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

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

# 2. Load the data
datamodule = SummarizationData.from_csv(
    "input",
    "target",
    train_file="data/xsum/train.csv",
    val_file="data/xsum/valid.csv",
    test_file="data/xsum/test.csv",
)

# 3. Build the model
model = SummarizationTask()

# 4. Create the trainer. Run once on data
trainer = Trainer(gpus=int(torch.cuda.is_available()), fast_dev_run=True)

# 5. Fine-tune the model
trainer.finetune(model, datamodule=datamodule)

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

To run the example:

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

9.4 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_csv(
    "input",
    "target",
    train_file="data/xsum/train.csv",
    val_file="data/xsum/valid.csv",
    test_file="data/xsum/test.csv",
```

(continues on next page)

(continued from previous page)

```
    backbone="google/mt5-small",
)

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

9.5 API reference

9.5.1 SummarizationTask

class flash.text.**SummarizationTask**(*backbone='sshleifer/tiny-mbart'*, *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

9.5.2 SummarizationData

class flash.text.**SummarizationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

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

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

Parameters

- **train_files** (Optional[Sequence[str]]) – A sequence of files to use as the train inputs.
- **train_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per train file) to use as the train targets.
- **val_files** (Optional[Sequence[str]]) – A sequence of files to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation file) to use as the validation targets.
- **test_files** (Optional[Sequence[str]]) – A sequence of files to use as the test inputs.
- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test file) to use as the test targets.
- **predict_files** (Optional[Sequence[str]]) – A sequence of files to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.

- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_files(
    train_files=["image_1.png", "image_2.png", "image_3.png"],
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

CHAPTER

TEN

TEXT CLASSIFICATION

10.1 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.

10.2 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.core.classification import Labels
from flash.core.data.utils 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")

model.serializer = Labels()

# 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!
```

(continues on next page)

(continued from previous page)

```
datamodule = TextClassificationData.from_csv(
    "review",
    predict_file="data/imdb/predict.csv",
)
predictions = Trainer().predict(model, datamodule=datamodule)
print(predictions)
```

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

10.3 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 to train our model!

```
import flash
from flash.core.data.utils 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_csv(
    train_file="data/imdb/train.csv",
    val_file="data/imdb/valid.csv",
    test_file="data/imdb/test.csv",
    input_fields="review",
    target_fields="sentiment",
    batch_size=16,
)

# 3. Build the model
model = TextClassifier(num_classes=datamodule.num_classes)

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

# 5. Fine-tune the model
trainer.finetune(model, datamodule=datamodule, strategy="freeze")

# 6. Test model
trainer.test(model)
```

(continues on next page)

(continued from previous page)

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

To run the example:

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

10.4 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_csv(
    "review",
    "sentiment",
    backbone="bert-base-chinese",
    train_file="data/imdb/train.csv",
    val_file="data/imdb/valid.csv",
    batch_size=512
)

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

10.5 API reference

10.5.1 TextClassifier

class flash.text.classification.model.**TextClassifier**(*num_classes*, *backbone='prajjwal1/bert-medium'*, *optimizer=torch.optim.Adam*, *metrics=None*, *learning_rate=0.01*, *multi_label=False*, *serializer=None*)

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*
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

step(*batch*, *batch_idx*)

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

Return type dict

10.5.2 TextClassificationData

class flash.text.classification.data.**TextClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

Data Module for text classification tasks

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

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

Parameters

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

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

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_files(
    train_files=["image_1.png", "image_2.png", "image_3.png"],
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

CHAPTER

ELEVEN

TABULAR CLASSIFICATION

11.1 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.

11.2 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:

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

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

```
from torchmetrics.classification import Accuracy, Precision, Recall

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

```

(continues on next page)

(continued from previous page)

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

# 2. Load the data
datamodule = TabularData.from_csv(
    ["Sex", "Age", "SibSp", "Parch", "Ticket", "Cabin", "Embarked"],
    "Fare",
    target_fields="Survived",
    train_file="./data/titanic/titanic.csv",
    test_file="./data/titanic/test.csv",
    val_split=0.25,
)

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

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

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

# 6. Test model
trainer.test(model)

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

11.3 Inference

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

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

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

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

model.serializer = Labels(['Did not survive', 'Survived'])

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

11.4 API reference

11.4.1 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=0.01*, *multi_label=False*, *serializer=None*, ***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*
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.
- ****tabnet_kwargs** – Optional additional arguments for the TabNet model, see pytorch_tabnet.

11.4.2 TabularData

class flash.tabular.**TabularData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

Data module for tabular tasks

classmethod TabularData.**from_csv**(*categorical_fields*, *numerical_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *is_regression=False*, ***preprocess_kwargs*)

Creates a TabularData object from the given CSV files.

Parameters

- **categorical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing categorical inputs.

- **numerical_fields** (Union[str, List[str], None]) – The field or fields (columns) in the CSV file containing numerical inputs.
- **target_fields** (Optional[str]) – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (Optional[str]) – The CSV file containing the training data.
- **val_file** (Optional[str]) – The CSV file containing the validation data.
- **test_file** (Optional[str]) – The CSV file containing the testing data.
- **predict_file** (Optional[str]) – The CSV file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **is_regression** (bool) – If True, targets will be formatted as floating point. If False, targets will be formatted as integers.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = TabularData.from_csv(
    "categorical_input",
    "numerical_input",
    "target",
    train_file="train_data.csv",
)
```

classmethod TabularData.**from_data_frame**(*categorical_fields*, *numerical_fields*, *target_fields=None*, *train_data_frame=None*, *val_data_frame=None*, *test_data_frame=None*, *predict_data_frame=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *is_regression=False*, ***preprocess_kwargs*)

Creates a `TabularData` object from the given data frames.

Parameters

- **categorical_fields** (`Union[str, List[str], None]`) – The field or fields (columns) in the CSV file containing categorical inputs.
- **numerical_fields** (`Union[str, List[str], None]`) – The field or fields (columns) in the CSV file containing numerical inputs.
- **target_fields** (`Optional[str]`) – The field or fields (columns) in the CSV file to use for the target.
- **train_data_frame** (`Optional[object]`) – The pandas `DataFrame` containing the training data.
- **val_data_frame** (`Optional[object]`) – The pandas `DataFrame` containing the validation data.
- **test_data_frame** (`Optional[object]`) – The pandas `DataFrame` containing the testing data.
- **predict_data_frame** (`Optional[object]`) – The pandas `DataFrame` containing the data to use when predicting.
- **train_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during training which maps *`Preprocess`* hook names to callable transforms.
- **val_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during validation which maps *`Preprocess`* hook names to callable transforms.
- **test_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during testing which maps *`Preprocess`* hook names to callable transforms.
- **predict_transform** (`Optional[Dict[str, Callable]]`) – The dictionary of transforms to use during predicting which maps *`Preprocess`* hook names to callable transforms.
- **data_fetcher** (`Optional[BaseDataFetcher]`) – The *`BaseDataFetcher`* to pass to the *`DataModule`*.
- **preprocess** (`Optional[Preprocess]`) – The `Preprocess` to pass to the *`DataModule`*. If `None`, `cls.preprocess_cls` will be constructed and used.
- **val_split** (`Optional[float]`) – The `val_split` argument to pass to the *`DataModule`*.
- **batch_size** (`int`) – The `batch_size` argument to pass to the *`DataModule`*.
- **num_workers** (`Optional[int]`) – The `num_workers` argument to pass to the *`DataModule`*.

- **is_regression** (bool) – If True, targets will be formatted as floating point. If False, targets will be formatted as integers.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = TabularData.from_data_frame(
    "categorical_input",
    "numerical_input",
    "target",
    train_data_frame=train_data,
)
```

CHAPTER

TWELVE

TRANSLATION

12.1 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.

12.2 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 using *TranslationTask* *predict* method:

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

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

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

# 3. Translate a few sentences!
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)
```

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

12.3 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 torch

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

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

backbone = "t5-small"

# 2. Load the data
datamodule = TranslationData.from_csv(
    "input",
    "target",
    train_file="data/wmt_en_ro/train.csv",
    val_file="data/wmt_en_ro/valid.csv",
    test_file="data/wmt_en_ro/test.csv",
    batch_size=1,
    backbone=backbone,
)

# 3. Build the model
model = TranslationTask(backbone=backbone)

# 4. Create the trainer
trainer = flash.Trainer(
    precision=16 if torch.cuda.is_available() else 32,
    gpus=int(torch.cuda.is_available()),
    fast_dev_run=True,
)

# 5. Fine-tune the model
trainer.finetune(model, datamodule=datamodule)

# 6. Test model
trainer.test(model)

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

To run the example:

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

12.4 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_csv(
    "input",
    "target",
    backbone="t5-small",
    train_file="data/wmt_en_ro/train.csv",
    val_file="data/wmt_en_ro/valid.csv",
    test_file="data/wmt_en_ro/test.csv",
)

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

12.5 API reference

12.5.1 TranslationTask

class flash.text.**TranslationTask**(*backbone='t5-small'*, *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

12.5.2 TranslationData

class flash.text.**TranslationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)
Data module for Translation tasks.

classmethod TranslationData.**from_files**(*train_files=None*, *train_targets=None*, *val_files=None*, *val_targets=None*, *test_files=None*, *test_targets=None*, *predict_files=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)
Creates a *DataModule* object from the given sequences of files using the *DataSource* of name *FILES* from the passed or constructed *Preprocess*.

Parameters

- **train_files**¶ (Optional[Sequence[str]]) – A sequence of files to use as the train inputs.
- **train_targets**¶ (Optional[Sequence[Any]]) – A sequence of targets (one per train file) to use as the train targets.
- **val_files**¶ (Optional[Sequence[str]]) – A sequence of files to use as the validation inputs.
- **val_targets**¶ (Optional[Sequence[Any]]) – A sequence of targets (one per validation file) to use as the validation targets.
- **test_files**¶ (Optional[Sequence[str]]) – A sequence of files to use as the test inputs.
- **test_targets**¶ (Optional[Sequence[Any]]) – A sequence of targets (one per test file) to use as the test targets.
- **predict_files**¶ (Optional[Sequence[str]]) – A sequence of files to use when predicting.
- **train_transform**¶ (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform**¶ (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.

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

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_files(
    train_files=["image_1.png", "image_2.png", "image_3.png"],
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

CHAPTER

THIRTEEN

OBJECT DETECTION

13.1 The task

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

13.2 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 import Trainer
from flash.core.data.utils import download_data
from flash.image import 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 model from a checkpoint
model = ObjectDetector.load_from_checkpoint("https://flash-weights.s3.amazonaws.com/
↪object_detection_model.pt")
```

(continues on next page)

(continued from previous page)

```
# 3. Detect the object on the images
predictions = model.predict([
    "data/coco128/images/train2017/000000000025.jpg",
    "data/coco128/images/train2017/000000000520.jpg",
    "data/coco128/images/train2017/000000000532.jpg",
])
print(predictions)
```

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

13.3 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.utils import download_data
from flash.image 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",
    val_split=0.3,
    batch_size=4,
    num_workers=4,
)

# 3. Build the model
model = ObjectDetector(model="retinanet", num_classes=datamodule.num_classes)

# 4. Create the trainer
trainer = flash.Trainer(max_epochs=3, limit_train_batches=1, limit_val_batches=1)

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

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

13.4 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.

13.5 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

13.6 API reference

13.6.1 ObjectDetector

class flash.image.**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.AdamW*, *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

13.6.2 ObjectDetectionData

class flash.image.**ObjectDetectionData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

classmethod ObjectDetectionData.**from_coco**(*train_folder=None*, *train_ann_file=None*, *val_folder=None*, *val_ann_file=None*, *test_folder=None*, *test_ann_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a `ObjectDetectionData` object from the given data folders and corresponding target folders.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **train_ann_file** (Optional[str]) – The COCO format annotation file.
- **val_folder** (Optional[str]) – The folder containing the validation data.

- **val_ann_file** (Optional[str]) – The COCO format annotation file.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_ann_file** (Optional[str]) – The COCO format annotation file.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Returns The constructed data module.

Examples:

```
data_module = SemanticSegmentationData.from_coco(
    train_folder="train_folder",
    train_ann_file="annotations.json",
)
```

CHAPTER

FOURTEEN

VIDEO CLASSIFICATION

14.1 The task

Typically, Video Classification refers to the task of producing a label for actions identified in a given video.

The task predicts which 'class' the video clip most likely belongs to with a degree of certainty.

A class is a label that describes what action is being performed within the video clip, such as **swimming** , **playing piano**, etc.

For example, we can train the video classifier task on video clips with human actions and it will learn to predict the probability that a video contains a certain human action.

Lightning Flash *VideoClassifier* and *VideoClassificationData* relies on PyTorchVideo internally.

You can use any models from PyTorchVideo Model Zoo with the *VideoClassifier*.

14.2 Finetuning

Let's say you wanted to develop a model that could determine whether a video clip contains a human **swimming** or **playing piano**, using the Kinetics dataset. Once we download the data using `download_data()`, all we need is the train data and validation data folders to create the *VideoClassificationData*.

```
video_dataset
├── train
│   ├── class_1
│   │   ├── a.ext
│   │   ├── b.ext
│   │   ...
│   └── class_n
│       ├── c.ext
│       ├── d.ext
│       ...
└── val
    ├── class_1
    │   ├── e.ext
    │   ├── f.ext
    │   ...
    └── class_n
        ├── g.ext
        ├── h.ext
        ...
```

```
import os
import sys
from typing import Callable, List

import torch
from torch.utils.data.sampler import RandomSampler

import flash
from flash.core.classification import Labels
from flash.core.data.utils import download_data
from flash.core.finetuning import NoFreeze
from flash.core.utilities.imports import _KORNIA_AVAILABLE, _PYTORCHVIDEO_AVAILABLE
from flash.video import VideoClassificationData, VideoClassifier

if _PYTORCHVIDEO_AVAILABLE and _KORNIA_AVAILABLE:
    import kornia.augmentation as K
    from pytorchvideo.transforms import ApplyTransformToKey, RandomShortSideScale,␣
↪UniformTemporalSubsample
    from torchvision.transforms import CenterCrop, Compose, RandomCrop,␣
↪RandomHorizontalFlip
else:
    print("Please, run `pip install torchvideo kornia`")
    sys.exit(1)

if __name__ == '__main__':

    # 1. Download a video clip dataset. Find more dataset at https://pytorchvideo.
↪readthedocs.io/en/latest/data.html
    download_data("https://pl-flash-data.s3.amazonaws.com/kinetics.zip")

    # 2. [Optional] Specify transforms to be used during training.
    # Flash helps you to place your transform exactly where you want.
    # Learn more at:
    # https://lightning-flash.readthedocs.io/en/latest/general/data.html#flash.core.
↪data.process.Preprocess
    post_tensor_transform = [UniformTemporalSubsample(8), RandomShortSideScale(min_
↪size=256, max_size=320)]
    per_batch_transform_on_device = [K.Normalize(torch.tensor([0.45, 0.45, 0.45]),␣
↪torch.tensor([0.225, 0.225, 0.225]))]

    train_post_tensor_transform = post_tensor_transform + [RandomCrop(244),␣
↪RandomHorizontalFlip(p=0.5)]
    val_post_tensor_transform = post_tensor_transform + [CenterCrop(244)]
    train_per_batch_transform_on_device = per_batch_transform_on_device

    def make_transform(
        post_tensor_transform: List[Callable] = post_tensor_transform,
        per_batch_transform_on_device: List[Callable] = per_batch_transform_on_device
    ):
        return {
            "post_tensor_transform": Compose([
                ApplyTransformToKey(
                    key="video",
                    transform=Compose(post_tensor_transform),
                ),
            ]),
            "per_batch_transform_on_device": Compose([
```

(continues on next page)

(continued from previous page)

```
                ApplyTransformToKey(
                    key="video",
                    transform=K.VideoSequential(
                        *per_batch_transform_on_device, data_format="BCTHW", same_on_
↪frame=False
                    )
                ),
            ]),
        }

    # 3. Load the data from directories.
    datamodule = VideoClassificationData.from_folders(
        train_folder=os.path.join(flash.PROJECT_ROOT, "data/kinetics/train"),
        val_folder=os.path.join(flash.PROJECT_ROOT, "data/kinetics/val"),
        predict_folder=os.path.join(flash.PROJECT_ROOT, "data/kinetics/predict"),
        train_transform=make_transform(train_post_tensor_transform),
        val_transform=make_transform(val_post_tensor_transform),
        predict_transform=make_transform(val_post_tensor_transform),
        batch_size=8,
        clip_sampler="uniform",
        clip_duration=1,
        video_sampler=RandomSampler,
        decode_audio=False,
        num_workers=8
    )

    # 4. List the available models
    print(VideoClassifier.available_backbones())
    # out: ['efficient_x3d_s', 'efficient_x3d_xs', ... ,slowfast_r50', 'x3d_m', 'x3d_s
↪', 'x3d_xs']
    print(VideoClassifier.get_backbone_details("x3d_xs"))

    # 5. Build the VideoClassifier with a PyTorchVideo backbone.
    model = VideoClassifier(
        backbone="x3d_xs", num_classes=datamodule.num_classes, serializer=Labels(),␣
↪pretrained=False
    )

    # 6. Finetune the model
    trainer = flash.Trainer(fast_dev_run=True)
    trainer.finetune(model, datamodule=datamodule, strategy=NoFreeze())

    trainer.save_checkpoint("video_classification.pt")

    # 7. Make a prediction
    predictions = model.predict(os.path.join(flash.PROJECT_ROOT, "data/kinetics/
↪predict"))
    print(predictions)
    # ['marching', 'flying_kite', 'archery', 'high_jump', 'bowling']
```

14.3 API reference

14.3.1 VideoClassifier

class flash.video.**VideoClassifier**(*num_classes*, *backbone='slow_r50'*, *backbone_kwargs=None*, *pretrained=True*, *loss_fn=torch.nn.functional.cross_entropy*, *optimizer=torch.optim.SGD*, *metrics=torchmetrics.Accuracy*, *learning_rate=0.001*, *head=None*, *serializer=None*)

Task that classifies videos.

Parameters

- **num_classes** (int) – Number of classes to classify.
- **backbone** (Union[str, Module]) – A string mapped to pytorch_video backbones or nn.Module, defaults to "slowfast_r50".
- **backbone_kwargs** (Optional[Dict]) – Arguments to customize the backbone from PyTorchVideo.
- **pretrained** (bool) – Use a pretrained backbone, defaults to True.
- **loss_fn** (Callable) – Loss function for training, defaults to torch.nn.functional.cross_entropy().
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to torch.optim.SGD.
- **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.

step(*batch*, *batch_idx*)

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

Return type Any

14.3.2 VideoClassificationData

class flash.video.**VideoClassificationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

Data module for Video classification tasks.

CHAPTER

FIFTEEN

SEMANTIC SEGMENTATION

15.1 The task

Semantic Segmentation, or image segmentation, is the task of performing classification at a pixel-level, meaning each pixel will associated to a given class. The model output shape is `(batch_size, num_classes, heigh, width)`.

See more: https://paperswithcode.com/task/semantic-segmentation

15.2 Inference

A *`SemanticSegmentation`* *fcn_resnet50* pre-trained on CARLA simulator is provided for the inference example.

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

```
from flash.core.data.utils import download_data
from flash.image import SemanticSegmentation
from flash.image.segmentation.serialization import SegmentationLabels

# 1. Download the data
# This is a Dataset with Semantic Segmentation Labels generated via CARLA self-
↪driving simulator.
# The data was generated as part of the Lyft Udacity Challenge.
# More info here: https://www.kaggle.com/kumaresanmanickavelu/lyft-udacity-challenge
download_data(
    "https://github.com/ongchinkiat/LyftPerceptionChallenge/releases/download/v0.1/
↪carla-capture-20180513A.zip", "data/"
)

# 2. Load the model from a checkpoint
model = SemanticSegmentation.load_from_checkpoint(
    "https://flash-weights.s3.amazonaws.com/semantic_segmentation_model.pt"
)
model.serializer = SegmentationLabels(visualize=True)

# 3. Predict what's on a few images and visualize!
predictions = model.predict([
    "data/CameraRGB/F61-1.png",
    "data/CameraRGB/F62-1.png",
    "data/CameraRGB/F63-1.png",
])
```

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

15.3 Finetuning

you now want to customise your model with new data using the same dataset. Once we download the data using `download_data()`, all we need is the train data and validation data folders to create the *SemanticSegmentationData*.

Note: the dataset is structured in a way that each sample (an image and its corresponding labels) is stored in separated directories but keeping the same filename.

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

Now all we need is to train our task!

```
import flash
from flash.core.data.utils import download_data
from flash.image import SemanticSegmentation, SemanticSegmentationData
from flash.image.segmentation.serialization import SegmentationLabels

# 1. Download the data
# This is a Dataset with Semantic Segmentation Labels generated via CARLA self-
↪driving simulator.
# The data was generated as part of the Lyft Udacity Challenge.
# More info here: https://www.kaggle.com/kumaresanmanickavelu/lyft-udacity-challenge
download_data(
    "https://github.com/ongchinkiat/LyftPerceptionChallenge/releases/download/v0.1/
↪carla-capture-20180513A.zip", "data/"
)

# 2.1 Load the data
datamodule = SemanticSegmentationData.from_folders(
    train_folder="data/CameraRGB",
    train_target_folder="data/CameraSeg",
    batch_size=4,
    val_split=0.3,
    image_size=(200, 200),  # (600, 800)
    num_classes=21,
)

# 2.2 Visualise the samples
datamodule.show_train_batch(["load_sample", "post_tensor_transform"])

# 3. Build the model
```

(continues on next page)

(continued from previous page)

```
model = SemanticSegmentation(
    backbone="torchvision/fcn_resnet50",
    num_classes=datamodule.num_classes,
    serializer=SegmentationLabels(visualize=True)
)

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

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

predictions = model.predict([
    "data/CameraRGB/F61-1.png",
    "data/CameraRGB/F62-1.png",
    "data/CameraRGB/F63-1.png",
])

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

15.4 API reference

15.4.1 SemanticSegmentation

class flash.image.**SemanticSegmentation**(*num_classes*, *backbone='torchvision/fcn_resnet50'*, *backbone_kwargs=None*, *pretrained=True*, *loss_fn=None*, *optimizer=torch.optim.AdamW*, *metrics=None*, *learning_rate=0.001*, *multi_label=False*, *serializer=None*, *postprocess=None*)

Task that performs semantic segmentation on images.

Use a built in backbone

Example:

```
from flash.image import SemanticSegmentation

segmentation = SemanticSegmentation(
    num_classes=21, backbone="torchvision/fcn_resnet50"
)
```

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 "torchvision/fcn_resnet50".

- **backbone_kwargs** (Optional[Dict]) – Additional arguments for the backbone configuration.
- **pretrained** (bool) – Use a pretrained backbone, defaults to False.
- **loss_fn** (Optional[Callable]) – Loss function for training, defaults to torch.nn.functional.cross_entropy().
- **optimizer** (Type[Optimizer]) – Optimizer to use for training, defaults to torch.optim.AdamW.
- **metrics** (Union[Callable, Mapping, Sequence, None]) – Metrics to compute for training and evaluation, defaults to torchmetrics.IoU.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.
- **multi_label** (bool) – Whether the targets are multi-label or not.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

postprocess_cls
alias of flash.image.segmentation.model.SemanticSegmentationPostprocess

15.4.2 SemanticSegmentationData

class flash.image.**SemanticSegmentationData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

Data module for semantic segmentation tasks.

classmethod SemanticSegmentationData.**from_folders**(*train_folder=None*, *train_target_folder=None*, *val_folder=None*, *val_target_folder=None*, *test_folder=None*, *test_target_folder=None*, *predict_folder=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, *num_classes=None*, *labels_map=None*, ***preprocess_kwargs*)

Creates a SemanticSegmentationData object from the given data folders and corresponding target folders.

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.

- **train_target_folder** (Optional[str]) – The folder containing the train targets (targets must have the same file name as their corresponding inputs).
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **val_target_folder** (Optional[str]) – The folder containing the validation targets (targets must have the same file name as their corresponding inputs).
- **test_folder** (Optional[str]) – The folder containing the test data.
- **test_target_folder** (Optional[str]) – The folder containing the test targets (targets must have the same file name as their corresponding inputs).
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **num_classes** (Optional[int]) – Number of classes within the segmentation mask.
- **labels_map** (Optional[Dict[int, Tuple[int, int, int]]]) – Mapping between a class_id and its corresponding color.
- **preprocess_kwargs** – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = SemanticSegmentationData.from_folders(
    train_folder="train_folder",
    train_target_folder="train_masks",
)
```

class flash.image.**SemanticSegmentationPreprocess**(*train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *image_size=(196, 196)*, *num_classes=None*, *labels_map=None*)

CHAPTER

SIXTEEN

STYLE TRANSFER

16.1 The task

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

Lightning Flash `StyleTransfer` and `StyleTransferData` internally rely on pystiche as backend.

16.2 Fit

First, you would have to import the `StyleTransfer` and `StyleTransferData` from Flash.

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

Then, download some content images and create a `StyleTransferData` DataModule.

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

data_module = StyleTransferData.from_folders(train_folder="data/coco128/images",␣
↪batch_size=4)
```

Select a style image and pass it to the *StyleTransfer* task.

```
style_image = pystiche.demo.images()["paint"].read(size=256)

model = StyleTransfer(style_image)
```

Finally, create a Flash *flash.core.trainer.Trainer* and pass it the model and datamodule.

```
trainer = flash.Trainer(max_epochs=2)
trainer.fit(model, data_module)
```

16.3 API reference

16.3.1 StyleTransfer

class flash.image.**StyleTransfer**(*style_image=None*, *model=None*, *backbone='vgg16'*, *content_layer='relu2_2'*, *content_weight=100000.0*, *style_layers=('relu1_2', 'relu2_2', 'relu3_3', 'relu4_3')*, *style_weight=10000000000.0*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *learning_rate=0.001*, *serializer=None*)

Task that transfer the style from an image onto another.

Example:

```
from flash.image.style_transfer import StyleTransfer

model = StyleTransfer(image_style)
```

Parameters

- **style_image** (Union[str, Tensor, None]) – Image or path to an image to derive the style from.
- **model** (Optional[Module]) – The model by the style transfer task.
- **backbone** (str) – A string or model to use to compute the style loss from.
- **content_layer** (str) – Which layer from the backbone to extract the content loss from.
- **content_weight** (float) – The weight associated with the content loss. A lower value will lose content over style.
- **style_layers** (Union[Sequence[str], str]) – Layers from the backbone to derive the style loss from.
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training the model.
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Optimizer keywords arguments.
- **scheduler** (Union[Type[LRScheduler], str, LRScheduler, None]) – Scheduler to use for training the model.
- **scheduler_kwargs** (Optional[Dict[str, Any]]) – Scheduler keywords arguments.
- **learning_rate** (float) – Learning rate to use for training, defaults to 1e-3.
- **serializer** (Union[*Serializer*, Mapping[str, *Serializer*], None]) – The *Serializer* to use when serializing prediction outputs.

16.3.2 StyleTransferData

class flash.image.**StyleTransferData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

CHAPTER

SEVENTEEN

MODEL

class flash.core.model.**Task**(*model=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=5e-05*, *preprocess=None*, *postprocess=None*, *serializer=None*)

A general Task.

Parameters

- **model** (Optional[Module]) – Model to use for the task.
- **loss_fn** (Union[Callable, Mapping, Sequence, None]) – Loss function for training
- **optimizer** (Union[Type[Optimizer], Optimizer]) – Optimizer to use for training, defaults to torch.optim.Adam.
- **metrics** (Union[Metric, Mapping, Sequence, None]) – Metrics to compute for training and evaluation.
- **learning_rate** (float) – Learning rate to use for training, defaults to 5e-5.
- **preprocess** (Optional[*Preprocess*]) – *Preprocess* to use as the default for this task.
- **postprocess** (Optional[*Postprocess*]) – *Postprocess* to use as the default for this task.

build_data_pipeline(*data_source=None*, *data_pipeline=None*)

Build a *DataPipeline* incorporating available *Preprocess* and *Postprocess* objects. These will be overridden in the following resolution order (lowest priority first):

- Lightning Datamodule, either attached to the *Trainer* or to the *Task*.
- *Task* defaults given to Task.__init__().
- *Task* manual overrides by setting *data_pipeline*.
- *DataPipeline* passed to this method.

Parameters **data_pipeline** (Optional[*DataPipeline*]) – Optional highest priority source of *Preprocess* and *Postprocess*.

Return type Optional[*DataPipeline*]

Returns The fully resolved *DataPipeline*.

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*

get_num_training_steps()
Total training steps inferred from datamodule and devices.

Return type int

predict(*x*, *data_source=None*, *data_pipeline=None*)
Predict function for raw data or processed data

Parameters

- **x**¶ (Any) – Input to predict. Can be raw data or processed data. If str, assumed to be a folder of data.
- **data_pipeline**¶ (Optional[*DataPipeline*]) – Use this to override the current data pipeline

Return type Any

Returns The post-processed model predictions

property serializer
The current *Serializer* associated with this model. If this property was set to a mapping (e.g. `.serializer = {'output1': SerializerOne()}`) then this will be a `MappingSerializer`.

Return type Optional[*Serializer*]

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

Return type Any

CHAPTER

EIGHTEEN

DATA

18.1 Terminology

Here are common terms you need to be familiar with:

Table 1: Terminology

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

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

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

18.3 Data Processing

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

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

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

Here are the primary advantages:

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

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

Check out *Preprocess* for some examples.

18.4 How to customize existing datamodules

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

```
from flash import Trainer
from flash.core.data.data_module import DataModule

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

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

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

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

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

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

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

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

class CustomImageClassificationPreprocess(ImageClassificationPreprocess):

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

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

18.5 Custom Preprocess + Datamodule

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

18.5.1 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:

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

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

trainer.fit(model, dm)
```

18.5.2 2. The DataSource

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

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

class ImageClassificationFoldersDataSource(DataSource):

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

        # metadata contains the image path and its corresponding label with the␣
↪following structure:
        # [(image_path_1, label_1), ... (image_path_n, label_n)].
        metadata = make_dataset(folder)

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

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

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

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

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

18.5.3 3. The Preprocess

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

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

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

    def __init__(
        self,
        train_transform: Optional[Dict[str, Callable]] = None,
        val_transform: Optional[Dict[str, Callable]] = None,
        test_transform: Optional[Dict[str, Callable]] = None,
        predict_transform: Optional[Dict[str, Callable]] = None,
    ):
        super().__init__(
            train_transform=train_transform,
            val_transform=val_transform,
            test_transform=test_transform,
            predict_transform=predict_transform,
            data_sources={
                DefaultDataSources.FOLDERS: ImageClassificationFoldersDataSource(),
            },
            default_data_source=DefaultDataSources.FOLDERS,
        )

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

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

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

18.5.4 4. The DataModule

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

```
from flash.core.data.data_module import DataModule

class ImageClassificationDataModule(DataModule):

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

18.6 API reference

18.6.1 DataSource

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

The DataSource class encapsulates two hooks: load_data and load_sample. The *to_datasets()* method can then be used to automatically construct data sets from the hooks.

generate_dataset(*data*, *running_stage*)

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

Parameters

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

Return type Union[AutoDataset, IterableAutoDataset, None]

Returns The constructed BaseAutoDataset.

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

Given the data argument, the load_data hook produces a sequence or iterable of samples or sample metadata. The data argument can be anything, but this method should return a sequence or iterable of mappings from string (e.g. "input", "target", "bbox", etc.) to data (e.g. a target value) or metadata (e.g. a filename). Where possible, any heavy data loading should be performed in *load_sample()*. If the output is an iterable rather than a sequence (that is, it doesn't have length) then the generated dataset will be an IterableDataset.

Parameters

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

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

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

Example:

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

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

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

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

Parameters

- **sample**¶ (Mapping[str, Any]) – An element (sample or sample metadata) from the output of a call to *load_data()*.

- **dataset** (Optional[Any]) – Overriding methods can optionally include the dataset argument. Any attributes set on the dataset (e.g. num_classes) will also be set on the generated dataset.

Return type Any

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

Example:

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

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

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

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

Parameters

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

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

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

class flash.core.data.data_source.**DefaultDataSources**(**args*, ***kwargs*)

The DefaultDataSources enum contains the data source names used by all of the default from_* methods in *DataModule*.

CSV = 'csv'

DATASET = 'dataset'

FILES = 'files'

FOLDERS = 'folders'

JSON = 'json'

NUMPY = 'numpy'

TENSORS = 'tensors'

class flash.core.data.data_source.**DefaultDataKeys**(**args*, ***kwargs*)

The DefaultDataKeys enum contains the keys that are used by built-in data sources to refer to inputs and targets.

INPUT = 'input'

```
METADATA = 'metadata'

PREDS = 'preds'

TARGET = 'target'
```

18.6.2 Preprocess

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

The *Preprocess* encapsulates all the data processing logic that should run before the data is passed to the model. It is particularly useful when you want to provide an end to end implementation which works with 4 different stages: `train`, `validation`, `test`, and inference (`predict`).

The *Preprocess* supports the following hooks:

- **`pre_tensor_transform`: Performs transforms on a single data sample.** Example:

```
* Input: Receive a PIL Image and its label.

* Action: Rotate the PIL Image.

* Output: Return the rotated PIL image and its label.
```

- **`to_tensor_transform`: Converts a single data sample to a tensor / data structure containing tensors.** Example:

```
* Input: Receive the rotated PIL Image and its label.

* Action: Convert the rotated PIL Image to a tensor.

* Output: Return the tensored image and its label.
```

- **`post_tensor_transform`: Performs transform on a single tensor sample.** Example:

```
* Input: Receive the tensored image and its label.

* Action: Flip the tensored image randomly.

* Output: Return the tensored image and its label.
```

- **`per_batch_transform`: Performs transforms on a batch.** In this example, we decided not to override the hook.

- **`per_sample_transform_on_device`: Performs transform on a sample already on a `GPU` or `TPU`.** Example:

```
* Input: Receive a tensored image on device and its label.

* Action: Apply random transforms.

* Output: Return an augmented tensored image on device and its label.
```

- **`collate`: Converts a sequence of data samples into a batch.** Defaults to `torch.utils.data._utils.collate.default_collate`. Example:

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

* Action: Collate the list of images into batch.

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

- **per_batch_transform_on_device: Performs transform on a batch already on GPU or TPU.**
  Example:

```
* Input: Receive a batch of images and their labels.

* Action: Apply normalization on the batch by subtracting the mean
    and dividing by the standard deviation from ImageNet.

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

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

Data processing can be configured by overriding hooks or through transforms. The preprocess transforms are given as a mapping from hook names to callables. Default transforms can be configured by overriding the `default_transforms` or `{train,val,test,predict}_default_transforms` methods. These can then be overridden by the user with the `{train,val,test,predict}_transform` arguments to the `Preprocess`. All of the hooks can be used in the transform mappings.

Example:

```
class CustomPreprocess(Preprocess):

    def default_transforms() -> Mapping[str, Callable]:
        return {
            "to_tensor_transform": transforms.ToTensor(),
            "collate": torch.utils.data._utils.collate.default_collate,
        }

    def train_default_transforms() -> Mapping[str, Callable]:
        return {
            "pre_tensor_transform": transforms.RandomHorizontalFlip(),
            "to_tensor_transform": transforms.ToTensor(),
            "collate": torch.utils.data._utils.collate.default_collate,
        }
```

When overriding hooks for particular stages, you can prefix with `train`, `val`, `test` or `predict`. For example, you can achieve the same as the above example by implementing `train_pre_tensor_transform` and `train_to_tensor_transform`.

Example:

```
class CustomPreprocess(Preprocess):

    def train_pre_tensor_transform(self, sample: PIL.Image) -> PIL.Image:
        return transforms.RandomHorizontalFlip()(sample)

    def to_tensor_transform(self, sample: PIL.Image) -> torch.Tensor:
```

(continues on next page)

(continued from previous page)

```
        return transforms.ToTensor()(sample)

    def collate(self, samples: List[torch.Tensor]) -> torch.Tensor:
        return torch.utils.data._utils.collate.default_collate(samples)
```

Each hook is aware of the Trainer running stage through booleans. These are useful for adapting functionality for a stage without duplicating code.

Example:

```
class CustomPreprocess(Preprocess):

    def pre_tensor_transform(self, sample: PIL.Image) -> PIL.Image:

        if self.training:
            # logic for training

        elif self.validating:
            # logic for validation

        elif self.testing:
            # logic for testing

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

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

Return type Sequence[str]

Returns The list of data source names.

collate(*samples*)
Transform to convert a sequence of samples to a collated batch.

Return type Any

data_source_of_name(*data_source_name*)
Get the *DataSource* of the given name from the *Preprocess*.

Parameters **data_source_name** (str) – The name of the data source to look up.

Return type *DataSource*

Returns The *DataSource* of the given name.

Raises **MisconfigurationException** – If the requested data source is not configured by this *Preprocess*.

default_transforms()
The default transforms to use. Will be overridden by transforms passed to the __init__.

Return type Optional[Dict[str, Callable]]

per_batch_transform(*batch*)
Transforms to apply to a whole batch (if possible use this for efficiency).

Note: This option is mutually exclusive with *per_sample_transform_on_device()*, since if

both are specified, uncollation has to be applied.

Return type Any

per_batch_transform_on_device(*batch*)
Transforms to apply to a whole batch (if possible use this for efficiency).

Note: This function won't be called within the dataloader workers, since to make that happen each of the workers would have to create it's own CUDA-context which would pollute GPU memory (if on GPU).

Return type Any

per_sample_transform_on_device(*sample*)
Transforms to apply to the data before the collation (per-sample basis).

Note: This option is mutually exclusive with *per_batch_transform()*, since if both are specified, uncollation has to be applied.

Note: This function won't be called within the dataloader workers, since to make that happen each of the workers would have to create it's own CUDA-context which would pollute GPU memory (if on GPU).

Return type Any

post_tensor_transform(*sample*)
Transforms to apply on a tensor.

Return type Tensor

pre_tensor_transform(*sample*)
Transforms to apply on a single object.

Return type Any

to_tensor_transform(*sample*)
Transforms to convert single object to a tensor.

Return type Tensor

property transforms
The transforms currently being used by this *Preprocess*.

Return type Dict[str, Optional[Dict[str, Callable]]]

18.6.3 Postprocess

class flash.core.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

18.6.4 Serializer

class flash.core.data.process.**Serializer**
A *Serializer* encapsulates a single `serialize` method which is used to convert the model ouptut into the desired output format when predicting.

disable()
Disable serialization.

enable()
Enable serialization.

serialize(*sample*)
Serialize the given sample into the desired output format.

Parameters **sample**¶ (Any) – The output from the *Postprocess*.

Return type Any

Returns The serialized output.

18.6.5 DataPipeline

class flash.core.data.data_pipeline.**DataPipeline**(*data_source=None*, *preprocess=None*, *postprocess=None*, *serializer=None*)

DataPipeline holds the engineering logic to connect *Preprocess* and/or *Postprocess* objects to the DataModule, Flash Task and Trainer.

Example:

```
class CustomPreprocess(Preprocess):
    pass

class CustomPostprocess(Postprocess):
    pass

custom_data_pipeline = DataPipeline(CustomPreprocess(), CustomPostprocess())

# And it can attached to both the datamodule and model.

datamodule.data_pipeline = custom_data_pipeline
model.data_pipeline = custom_data_pipeline
```

initialize(*data_pipeline_state=None*)

Creates the DataPipelineState and gives the reference to the: *Preprocess*, *Postprocess*, and *Serializer*. Once this has been called, any attempt to add new state will give a warning.

Return type DataPipelineState

18.6.6 DataModule

class flash.core.data.data_module.**DataModule**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)

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

Parameters

- **train_dataset** (Optional[Dataset]) – Dataset for training. Defaults to None.
- **val_dataset** (Optional[Dataset]) – Dataset for validating model performance during training. Defaults to None.
- **test_dataset** (Optional[Dataset]) – Dataset to test model performance. Defaults to None.
- **predict_dataset** (Optional[Dataset]) – Dataset for predicting. Defaults to None.
- **data_source** (Optional[*DataSource*]) – The *DataSource* that was used to create the datasets.
- **preprocess** (Optional[*Preprocess*]) – The *Preprocess* to use when constructing the *DataPipeline*. If None, a DefaultPreprocess will be used.

- **postprocess** (Optional[*Postprocess*]) – The *Postprocess* to use when constructing the *DataPipeline*. If None, a plain *Postprocess* will be used.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to attach to the *Preprocess*. If None, the output from *configure_data_fetcher()* will be used.
- **val_split** (Optional[float]) – An optional float which gives the relative amount of the training dataset to use for the validation dataset.
- **batch_size** (int) – The batch size to be used by the DataLoader. Defaults to 1.
- **num_workers** (Optional[int]) – The number of workers to use for parallelized loading. Defaults to None which equals the number of available CPU threads, or 0 for Windows or Darwin platform.

available_data_sources()

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

Return type Sequence[str]

Returns The list of data source names.

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

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

Return type *BaseDataFetcher*

classmethod from_csv(*input_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

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

Parameters

- **input_fields** (Union[str, Sequence[str]]) – The field or fields (columns) in the CSV file to use for the input.
- **target_fields** (Union[str, Sequence[str], None]) – The field or fields (columns) in the CSV file to use for the target.
- **train_file** (Optional[str]) – The CSV file containing the training data.
- **val_file** (Optional[str]) – The CSV file containing the validation data.
- **test_file** (Optional[str]) – The CSV file containing the testing data.
- **predict_file** (Optional[str]) – The CSV file containing the data to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.

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

Return type *DataModule*

Returns The constructed data module.

Examples:

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

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

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

Parameters

- **data_source** (str) – The name of the data source to use for the *load_data()*.
- **train_data** (Optional[Any]) – The input to *load_data()* to use when creating the train dataset.
- **val_data** (Optional[Any]) – The input to *load_data()* to use when creating the validation dataset.
- **test_data** (Optional[Any]) – The input to *load_data()* to use when creating the test dataset.
- **predict_data** (Optional[Any]) – The input to *load_data()* to use when creating the predict dataset.

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

Return type *DataModule*

Returns The constructed data module.

Examples:

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

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

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

Parameters

- **train_dataset** (Optional[Dataset]) – Dataset used during training.
- **val_dataset** (Optional[Dataset]) – Dataset used during validating.
- **test_dataset** (Optional[Dataset]) – Dataset used during testing.
- **predict_dataset** (Optional[Dataset]) – Dataset used during predicting.

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

Return type *DataModule*

Returns The constructed data module.

Examples:

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

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

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

Parameters

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

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

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_files(
    train_files=["image_1.png", "image_2.png", "image_3.png"],
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

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

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

Parameters

- **train_folder** (Optional[str]) – The folder containing the train data.
- **val_folder** (Optional[str]) – The folder containing the validation data.
- **test_folder** (Optional[str]) – The folder containing the test data.
- **predict_folder** (Optional[str]) – The folder containing the predict data.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.
- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

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

classmethod from_json(*input_fields*, *target_fields=None*, *train_file=None*, *val_file=None*, *test_file=None*, *predict_file=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

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

Parameters

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

Return type *DataModule*

Returns The constructed data module.

Examples:

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

classmethod from_numpy(*train_data=None*, *train_targets=None*, *val_data=None*, *val_targets=None*, *test_data=None*, *test_targets=None*, *predict_data=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

Creates a *DataModule* object from the given numpy array using the *DataSource* of name *NUMPY* from the passed or constructed *Preprocess*.

Parameters

- **train_data** (Optional[Collection[ndarray]]) – A numpy array to use as the train inputs.
- **train_targets** (Optional[Collection[Any]]) – A sequence of targets (one per train input) to use as the train targets.
- **val_data** (Optional[Collection[ndarray]]) – A numpy array to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation input) to use as the validation targets.
- **test_data** (Optional[Collection[ndarray]]) – A numpy array to use as the test inputs.
- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test input) to use as the test targets.
- **predict_data** (Optional[Collection[ndarray]]) – A numpy array to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.
- **val_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during validation which maps *Preprocess* hook names to callable transforms.
- **test_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during testing which maps *Preprocess* hook names to callable transforms.
- **predict_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during predicting which maps *Preprocess* hook names to callable transforms.
- **data_fetcher** (Optional[*BaseDataFetcher*]) – The *BaseDataFetcher* to pass to the *DataModule*.
- **preprocess** (Optional[*Preprocess*]) – The Preprocess to pass to the *DataModule*. If None, cls.preprocess_cls will be constructed and used.

- **val_split** (Optional[float]) – The val_split argument to pass to the *DataModule*.
- **batch_size** (int) – The batch_size argument to pass to the *DataModule*.
- **num_workers** (Optional[int]) – The num_workers argument to pass to the *DataModule*.
- **preprocess_kwargs** (Any) – Additional keyword arguments to use when constructing the preprocess. Will only be used if preprocess = None.

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_numpy(
    train_files=np.random.rand(3, 128),
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

classmethod from_tensors(*train_data=None*, *train_targets=None*, *val_data=None*, *val_targets=None*, *test_data=None*, *test_targets=None*, *predict_data=None*, *train_transform=None*, *val_transform=None*, *test_transform=None*, *predict_transform=None*, *data_fetcher=None*, *preprocess=None*, *val_split=None*, *batch_size=4*, *num_workers=None*, ***preprocess_kwargs*)

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

Parameters

- **train_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the train inputs.
- **train_targets** (Optional[Collection[Any]]) – A sequence of targets (one per train input) to use as the train targets.
- **val_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the validation inputs.
- **val_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per validation input) to use as the validation targets.
- **test_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use as the test inputs.
- **test_targets** (Optional[Sequence[Any]]) – A sequence of targets (one per test input) to use as the test targets.
- **predict_data** (Optional[Collection[Tensor]]) – A tensor or collection of tensors to use when predicting.
- **train_transform** (Optional[Dict[str, Callable]]) – The dictionary of transforms to use during training which maps *Preprocess* hook names to callable transforms.

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

Return type *DataModule*

Returns The constructed data module.

Examples:

```
data_module = DataModule.from_tensors(
    train_files=torch.rand(3, 128),
    train_targets=[1, 0, 1],
    train_transform={
        "to_tensor_transform": torch.as_tensor,
    },
)
```

postprocess_cls
alias of *flash.core.data.process.Postprocess*

property predict_dataset
This property returns the predict dataset

Return type Optional[Dataset]

preprocess_cls
alias of flash.core.data.process.DefaultPreprocess

show_predict_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the predict dataloader.

Return type None

show_test_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the test dataloader.

Return type None

show_train_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the train dataloader.

Return type None

show_val_batch(*hooks_names='load_sample'*, *reset=True*)
This function is used to visualize a batch from the validation dataloader.

Return type None

property test_dataset
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]

18.7 How it works behind the scenes

18.7.1 DataSource

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

Here is the `AutoDataset` pseudo-code.

Example:

```
class AutoDataset
    def __init__(
        self,
        data: List[Any],  # The result of a call to DataSource.load_data
        data_source: DataSource,
        running_stage: RunningStage,
    ) -> None:

        self.data = data
        self.data_source = data_source

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

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

18.7.2 Preprocess

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

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

    samples = type(samples)(samples)

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

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

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

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

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

Example:

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

    # 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)
```

18.7.3 Postprocess and Serializer

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

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

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

Here is the pseudo-code:

Example:

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

    batch = per_batch_transform(batch)

    samples = uncollate(batch)

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

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

CHAPTER

NINETEEN

CALLBACK

19.1 Flash Callback

FlashCallback is an extension of pytorch_lightning.callbacks.Callback.

A callback is a self-contained program that can be reused across projects.

Flash and Lightning have a callback system to execute callbacks when needed.

Callbacks should capture any NON-ESSENTIAL logic that is NOT required for your lightning module to run.

Same as PyTorch Lightning, Callbacks can be provided directly to the Trainer.

Example:

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

19.2 Available Callbacks

19.2.1 BaseDataFetcher

class flash.core.data.callback.**BaseDataFetcher**(*enabled=False*)

This class is used to profile *Preprocess* hook outputs.

By default, the callback won't profile the data being processed as it may lead to OOMError.

Example:

```
from flash.core.data.callback import BaseDataFetcher
from flash.core.data.data_module import DataModule
from flash.core.data.data_source import DataSource
from flash.core.data.process import Preprocess

class CustomPreprocess(Preprocess):

    def __init__(**kwargs):
        super().__init__(
            data_sources = {"inputs": DataSource()},
            **kwargs,
        )

class PrintData(BaseDataFetcher):
```

(continues on next page)

(continued from previous page)

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

class CustomDataModule(DataModule):

    preprocess_cls = CustomPreprocess

    @staticmethod
    def configure_data_fetcher():
        return PrintData()

    @classmethod
    def from_inputs(
        cls,
        train_data: Any,
        val_data: Any,
        test_data: Any,
        predict_data: Any,
    ) -> "CustomDataModule":
        return cls.from_data_source(
            "inputs",
            train_data=train_data,
            val_data=val_data,
            test_data=test_data,
            predict_data=predict_data,
            batch_size=5,
        )

dm = CustomDataModule.from_inputs(range(5), range(5), range(5), range(5))
data_fetcher = dm.data_fetcher

# By default, the ``data_fetcher`` is disabled to prevent OOM.
# The ``enable`` context manager will activate it.
with data_fetcher.enable():

    # This will fetch the first val dataloader batch.
    _ = next(iter(dm.val_dataloader()))

data_fetcher.print()
# out:
{
    'train': {},
    'test': {},
    'val': {
        'load_sample': [0, 1, 2, 3, 4],
        'pre_tensor_transform': [0, 1, 2, 3, 4],
        'to_tensor_transform': [0, 1, 2, 3, 4],
        'post_tensor_transform': [0, 1, 2, 3, 4],
        'collate': [tensor([0, 1, 2, 3, 4])],
        'per_batch_transform': [tensor([0, 1, 2, 3, 4])]},
    'predict': {}
}
data_fetcher.reset()
data_fetcher.print()
# out:
{
    'train': {},
```

(continues on next page)

(continued from previous page)

```
    'test': {},
    'val': {},
    'predict': {}
}
```

enable()
This function is used to enable to BaseDataFetcher

19.2.2 BaseVisualization

class flash.core.data.base_viz.**BaseVisualization**(*enabled=False*)
This Base Class is used to create visualization tool on top of *Preprocess* hooks.

Override any of the `show_{preprocess_hook_name}` to receive the associated data and visualize them.

Example:

```
from flash.image import ImageClassificationData
from flash.core.data.base_viz import BaseVisualization

class CustomBaseVisualization(BaseVisualization):

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

    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
```

(continues on next page)

(continued from previous page)

```
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*, *func_names_list*)
Override this function when you want to visualize a composition.

Return type None

show_collate(*batch*, *running_stage*)
Override to visualize preprocess `collate` output data.

Return type None

show_load_sample(*samples*, *running_stage*)
Override to visualize preprocess `load_sample` output data.

show_per_batch_transform(*batch*, *running_stage*)
Override to visualize preprocess `per_batch_transform` output data.

Return type None

show_per_batch_transform_on_device(*batch*, *running_stage*)
Override to visualize preprocess `per_batch_transform_on_device` output data.

Return type None

show_per_sample_transform_on_device(*samples*, *running_stage*)
Override to visualize preprocess `per_sample_transform_on_device` output data.

Return type None

show_post_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `post_tensor_transform` output data.

show_pre_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `pre_tensor_transform` output data.

show_to_tensor_transform(*samples*, *running_stage*)
Override to visualize preprocess `to_tensor_transform` output data.

19.3 API reference

19.3.1 FlashCallback

class flash.core.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

TWENTY

REGISTRY

20.1 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.

20.1.1 1. Imports

```
from functools import partial

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

20.1.2 2. Init a Registry

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

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

    backbones = FlashRegistry("backbones")

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

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

20.1.3 3. Adding new functions

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

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

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

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

20.1.4 4. Accessing registered functions

You can now access your function from your task!

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

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

Here's the output:

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

20.1.5 5. Pre-registered backbones

Flash provides populated registries containing lots of available backbones.

Example:

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

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

20.2 Flash Registry

20.2.1 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

TWENTYONE

TRAINING FROM SCRATCH

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

From the *Quick Start* guide.

To train a task from scratch:

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

Here's an example:

```
from pytorch_lightning import seed_everything

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

# set the random seeds.
seed_everything(42)

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

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

# 2. Build the model using desired Task
```

(continues on next page)

(continued from previous page)

```
model = ImageClassifier(backbone="resnet18", num_classes=datamodule.num_classes,␣
↪pretrained=False)

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

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

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

21.1 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.

21.1.1 Trainer API

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

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.

  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

TWENTYTWO

FINETUNING

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

22.1 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

22.2 Finetuning in Flash

From the *Quick Start* guide.

To use a Task for finetuning:

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

Here's an example of finetuning.

```
from pytorch_lightning import seed_everything

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

# set the random seeds.
seed_everything(42)

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

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

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

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

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

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

22.2.1 Using a finetuned model

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

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

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

We get the following output:

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

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

```
from flash.image import ImageClassifier

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

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

22.3 Finetune strategies

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

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

22.3.1 no_freeze

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

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

In pseudocode, this looks like:

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

backbone.unfreeze()
head.unfreeze()

train(backbone, head)
```

22.3.2 freeze

The freeze strategy keeps the backbone frozen throughout.

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

The pseudocode looks like:

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

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

train(backbone, head)
```

22.4 Advanced strategies

Every finetune strategy can also be customized.

22.4.1 freeze_unfreeze

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

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

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

```
from flash.core.finetuning import FreezeUnfreeze

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

Under the hood, the pseudocode looks like:

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

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

train(backbone, head, epochs=10)

# unfreeze after 10 epochs
backbone.unfreeze()

train(backbone, head)
```

22.4.2 unfreeze_milestones

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

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

```
from flash.core.finetuning import UnfreezeMilestones

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

Under the hood, the pseudocode looks like:

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

```

(continues on next page)

(continued from previous page)

```
# 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()
```

22.5 Custom Strategy

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

```
from flash.core.finetuning import FlashBaseFinetuning

# Create a finetuning callback
class FeatureExtractorFreezeUnfreeze(FlashBaseFinetuning):

    def __init__(self, unfreeze_epoch: int = 5, train_bn: bool = True):
        # this will set self.attr_names as ["backbone"]
        super().__init__("backbone", train_bn)
        self._unfreeze_epoch = unfreeze_epoch

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

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

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

CHAPTER

TWENTYTHREE

PREDICTIONS (INFERENCE)

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

23.1 Predict on a single sample of data

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

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


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

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

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

23.2 Predict on a csv file

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

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

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

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

23.3 Serializing predictions

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

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


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

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

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

# 4. Predict whether the image contains an ant or a bee
predictions = model.predict("data/hymenoptera_data/val/bees/65038344_52a45d090d.jpg")
print(predictions)
# out: [[0.5926494598388672, 0.40735048055648804]]
```

23.3.1 Classification serializers - API reference

Logits

class flash.core.classification.**Logits**(*multi_label=False*)

A *Serializer* which simply converts the model outputs (assumed to be logits) to a list.

Probabilities

class flash.core.classification.**Probabilities**(*multi_label=False*)

A *Serializer* which applies a softmax to the model outputs (assumed to be logits) and converts to a list.

Classes

class flash.core.classification.**Classes**(*multi_label=False*, *threshold=0.5*)

A *Serializer* which applies an argmax to the model outputs (either logits or probabilities) and converts to a list.

Parameters

- **multi_label** (`bool`) – If true, treats outputs as multi label logits.
- **threshold** (`float`) – The threshold to use for multi_label classification.

Labels

class flash.core.classification.**Labels**(*labels=None*, *multi_label=False*, *threshold=0.5*)

A *Serializer* which converts the model outputs (either logits or probabilities) to the label of the argmax classification.

Parameters

- **labels** (Optional[List[str]]) – A list of labels, assumed to map the class index to the label for that class. If labels is not provided, will attempt to get them from the LabelsState.
- **multi_label** (bool) – If true, treats outputs as multi label logits.
- **threshold** (float) – The threshold to use for multi_label classification.

CHAPTER

TWENTYFOUR

INTRODUCTION / SET-UP

24.1 Welcome

Before you begin, we'd like to express our gratitude to you for wanting to add a task to Flash. With Flash our aim is to create a great user experience, enabling awesome advanced applications with just a few lines of code. We're really pleased with what we've achieved with Flash and we hope you will be too. Now let's dive in!

24.2 Set-up

The Task template is designed to guide you through contributing a task to Flash. It contains the code, tests, and examples for a task that performs classification with a multi-layer perceptron, intended for use with the classic data sets from scikit-learn. The Flash tasks are organized in folders by data-type (image, text, video, etc.), with sub-folders for different task types (classification, regression, etc.).

Copy the files in flash/template/classification to a new sub-directory under the relevant data-type. If a data-type folder already exists for your task, then a task type sub-folder should be added containing the template files. If a data-type folder doesn't exist, then you will need to add that too. You should also copy the files from tests/template/classification to the corresponding data-type, task type folder in `tests`. For example, if you were adding an image classification task, you would do:

```
mkdir flash/image/classification
cp flash/template/classification/* flash/image/classification/
mkdir tests/image/classification
cp tests/template/classification/* tests/image/classification/
```

24.3 Tutorials

The tutorials in this section will walk you through all of the components you need to implement (or adapt from the template) for your custom task.

- *The Data*: our first tutorial goes over the best practices for implementing everything you need to connect data to your task
- *The Backbones*: the second tutorial shows you how to create an extensible backbone registry for your task
- *The Task*: now that we have the data and the models, in this tutorial we create our custom task
- *Optional Extras*: this tutorial covers some optional extras you can add if needed for your particular task
- *The Examples*: this tutorial guides you through creating some simple examples showing your task in action

- *The Tests*: in this tutorial, we cover best practices for writing some tests for your new task
- *The Docs*: in our final tutorial, we provide a template for you to create the docs page for your task

CHAPTER

TWENTYFIVE

THE DATA

The first step to contributing a task is to implement the classes we need to load some data. Inside data.py you should implement:

1. some *DataSource* classes *(optional)*
2. a *Preprocess*
3. a *DataModule*
4. a *BaseVisualization* *(optional)*
5. a *Postprocess* *(optional)*

25.1 DataSource

The *DataSource* class contains the logic for data loading from different sources such as folders, files, tensors, etc. Every Flash *DataModule* can be instantiated with *from_datasets()*. For each additional way you want the user to be able to instantiate your *DataModule*, you'll need to create a *DataSource*. Each *DataSource* has 2 methods:

- *load_data()* takes some dataset metadata (e.g. a folder name) as input and produces a sequence or iterable of samples or sample metadata.
- *load_sample()* then takes as input a single element from the output of `load_data` and returns a sample.

By default these methods just return their input, so you don't need both a *load_data()* and a *load_sample()* to create a *DataSource*. Where possible, you should override one of our existing *DataSource* classes.

Let's start by implementing a `TemplateNumpyDataSource`, which overrides `NumpyDataSource`. The main *DataSource* method that we have to implement is *load_data()*. As we're extending the `NumpyDataSource`, we expect the same `data` argument (in this case, a tuple containing data and corresponding target arrays).

We can also take the dataset argument. Any attributes we set on `dataset` will be available on the `Dataset` generated by our *DataSource*. In this data source, we'll set the `num_features` attribute.

Here's the code for our `TemplateNumpyDataSource.load_data` method:

```
def load_data(self, data: Tuple[np.ndarray, Sequence[Any]], dataset: Any) ->␣
↪Sequence[Mapping[str, Any]]:
    """Sets the ``num_features`` attribute and calls ``super().load_data``.

    Args:
        data: The tuple of ``np.ndarray`` (num_examples x num_features) and␣
↪associated targets.
        dataset: The object that we can set attributes (such as ``num_features``) on.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        A sequence of samples / sample metadata.
    """
    dataset.num_features = data[0].shape[1]
    return super().load_data(data, dataset)
```

Note: Later, when we add *our DataModule implementation*, we'll make `num_features` available to the user.

Sometimes you need to something a bit more custom. When creating a custom *`DataSource`*, the type of the `data` argument is up to you. For our template `Task`, it would be cool if the user could provide a scikit-learn `Bunch` as the data source. To achieve this, we'll add a `TemplateSKLearnDataSource` whose `load_data` expects a `Bunch` as input. We override our `TemplateNumpyDataSource` so that we can call `super` with the data and targets extracted from the `Bunch`. We perform two additional steps here to improve the user experience:

1. We set the `num_classes` attribute on the `dataset`. If `num_classes` is set, it is automatically made available as a property of the *`DataModule`*.
2. We create and set a `LabelsState`. The labels provided here will be shared with the *`Labels`* serializer, so the user doesn't need to provide them.

Here's the code for the `TemplateSKLearnDataSource.load_data` method:

```
def load_data(self, data: Bunch, dataset: Any) -> Sequence[Mapping[str, Any]]:
    """Gets the ``data`` and ``target`` attributes from the ``Bunch`` and passes them␣
↪to ``super().load_data``.

    Args:
        data: The scikit-learn data ``Bunch``.
        dataset: The object that we can set attributes (such as ``num_classes``) on.

    Returns:
        A sequence of samples / sample metadata.
    """
    dataset.num_classes = len(data.target_names)
    self.set_state(LabelsState(data.target_names))
    return super().load_data((data.data, data.target), dataset=dataset)
```

We can customize the behaviour of our *`load_data()`* for different stages, by prepending *train*, *val*, *test*, or *predict*. For our `TemplateSKLearnDataSource`, we don't want to provide any targets to the model when predicting. We can implement `predict_load_data` like this:

```
def predict_load_data(self, data: Bunch) -> Sequence[Mapping[str, Any]]:
    """Avoid including targets when predicting.

    Args:
        data: The scikit-learn data ``Bunch``.

    Returns:
        A sequence of samples / sample metadata.
    """
    return super().predict_load_data(data.data)
```

25.1.1 DataSource vs Dataset

A *DataSource* is not the same as a torch.utils.data.Dataset. When a from_* method is called on your *DataModule*, it gets the *DataSource* to use from the *Preprocess*. A Dataset is then created from the *DataSource* for each stage (*train*, *val*, *test*, *predict*) using the provided metadata (e.g. folder name, numpy array etc.).

The output of the *load_data()* can just be a torch.utils.data.Dataset instance. If the library that your Task is based on provides a custom dataset, you don't need to re-write it as a *DataSource*. For example, the *load_data()* of the VideoClassificationPathsDataSource just creates an EncodedVideoDataset from the given folder. Here's how it looks (from video/classification.data.py):

```
def load_data(self, data: str, dataset: Optional[Any] = None) -> 'EncodedVideoDataset
↪':
    ds: EncodedVideoDataset = labeled_encoded_video_dataset(
        pathlib.Path(data),
        self.clip_sampler,
        video_sampler=self.video_sampler,
        decode_audio=self.decode_audio,
        decoder=self.decoder,
    )
    if self.training:
        label_to_class_mapping = {p[1]: p[0].split("/")[-2] for p in ds._labeled_
↪videos._paths_and_labels}
        self.set_state(LabelsState(label_to_class_mapping))
        dataset.num_classes = len(np.unique([s[1]['label'] for s in ds._labeled_
↪videos]))
    return ds
```

25.2 Preprocess

The *Preprocess* object contains all the data transforms. Internally we inject the *Preprocess* transforms at several points along the pipeline.

Defining the standard transforms (typically at least a to_tensor_transform should be defined) for your *Preprocess* is as simple as implementing the default_transforms method. The *Preprocess* must take train_transform, val_transform, test_transform, and predict_transform arguments in the __init__. These arguments can be provided by the user (when creating the *DataModule*) to override the default transforms. Any additional arguments are up to you.

Inside the __init__, we make a call to super. This is where we register our data sources. Data sources should be given as a dictionary which maps data source name to data source object. The name can be anything, but if you want to take advantage of our built-in from_* classmethods, you should use *DefaultDataSources* as the names. In our case, we have both a *NUMPY* and a custom scikit-learn data source (which we'll call *"sklearn"*).

You should also provide a default_data_source. This is the name of the data source to use by default when predicting. It'd be cool if we could get predictions just from a numpy array, so we'll use *NUMPY* as the default.

Here's our TemplatePreprocess.__init__:

```
def __init__(
    self,
    train_transform: Optional[Dict[str, Callable]] = None,
    val_transform: Optional[Dict[str, Callable]] = None,
    test_transform: Optional[Dict[str, Callable]] = None,
    predict_transform: Optional[Dict[str, Callable]] = None,
```

(continues on next page)

(continued from previous page)

```
):
    super().__init__(
        train_transform=train_transform,
        val_transform=val_transform,
        test_transform=test_transform,
        predict_transform=predict_transform,
        data_sources={
            DefaultDataSources.NUMPY: TemplateNumpyDataSource(),
            "sklearn": TemplateSKLearnDataSource(),
        },
        default_data_source=DefaultDataSources.NUMPY,
    )
```

For our `TemplatePreprocess`, we'll just configure a default `to_tensor_transform`. Let's first define the transform as a `staticmethod`:

```
@staticmethod
def input_to_tensor(input: np.ndarray):
    """Transform which creates a tensor from the given numpy ``ndarray`` and converts␣
↪it to ``float``"""
    return torch.from_numpy(input).float()
```

Our inputs samples will be dictionaries whose keys are in the *DefaultDataKeys*. You can map each key to different transforms using `ApplyToKeys`. Here's our `default_transforms` method:

```
def default_transforms(self) -> Optional[Dict[str, Callable]]:
    """Configures the default ``to_tensor_transform``.

    Returns:
        Our dictionary of transforms.
    """
    return {
        "to_tensor_transform": nn.Sequential(
            ApplyToKeys(DefaultDataKeys.INPUT, self.input_to_tensor),
            ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
        ),
    }
```

25.3 DataModule

The *DataModule* is responsible for creating the `DataLoader` and injecting the transforms for each stage. When the user calls a `from_*` method (such as *from_numpy()*), the following steps take place:

1. The *from_data_source()* method is called with the name of the *DataSource* to use and the inputs to provide to *load_data()* for each stage.
2. The *Preprocess* is created from `cls.preprocess_cls` (if it wasn't provided by the user) with any provided transforms.
3. The *DataSource* of the provided name is retrieved from the *Preprocess*.
4. A `BaseAutoDataset` is created from the *DataSource* for each stage.
5. The *DataModule* is instantiated with the data sets.

To create our TemplateData *DataModule*, we first need to attach out preprocess class like this:

```
preprocess_cls = TemplatePreprocess
```

Since we provided a *NUMPY DataSource* in the TemplatePreprocess, *from_numpy()* will now work with our TemplateData.

If you've defined a fully custom *DataSource* (like our TemplateSKLearnDataSource), then you will need to write a from_* method for each. Here's the from_sklearn method for our TemplateData:

```
@classmethod
def from_sklearn(
    cls,
    train_bunch: Optional[Bunch] = None,
    val_bunch: Optional[Bunch] = None,
    test_bunch: Optional[Bunch] = None,
    predict_bunch: Optional[Bunch] = None,
    train_transform: Optional[Dict[str, Callable]] = None,
    val_transform: Optional[Dict[str, Callable]] = None,
    test_transform: Optional[Dict[str, Callable]] = None,
    predict_transform: Optional[Dict[str, Callable]] = None,
    data_fetcher: Optional[BaseDataFetcher] = None,
    preprocess: Optional[Preprocess] = None,
    val_split: Optional[float] = None,
    batch_size: int = 4,
    num_workers: Optional[int] = None,
    **preprocess_kwargs: Any,
):
    """This is our custom ``from_*`` method. It expects scikit-learn ``Bunch``␣
↪objects as input and passes them
    through to the :meth:`~flash.core.data.data_module.DataModule.from_data_source`␣
↪method underneath.

    Args:
        train_bunch: The scikit-learn ``Bunch`` containing the train data.
        val_bunch: The scikit-learn ``Bunch`` containing the validation data.
        test_bunch: The scikit-learn ``Bunch`` containing the test data.
        predict_bunch: The scikit-learn ``Bunch`` containing the predict data.
        train_transform: The dictionary of transforms to use during training which␣
↪maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        val_transform: The dictionary of transforms to use during validation which␣
↪maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        test_transform: The dictionary of transforms to use during testing which maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        predict_transform: The dictionary of transforms to use during predicting␣
↪which maps
            :class:`~flash.core.data.process.Preprocess` hook names to callable␣
↪transforms.
        data_fetcher: The :class:`~flash.core.data.callback.BaseDataFetcher` to pass␣
↪to the
```

(continues on next page)

(continued from previous page)

```
            :class:`~flash.core.data.data_module.DataModule`.
        preprocess: The :class:`~flash.core.data.data.Preprocess` to pass to the
            :class:`~flash.core.data.data_module.DataModule`. If ``None``, ``cls.
↪preprocess_cls`` will be
            constructed and used.
        val_split: The ``val_split`` argument to pass to the :class:`~flash.core.data.
↪data_module.DataModule`.
        batch_size: The ``batch_size`` argument to pass to the :class:`~flash.core.
↪data.data_module.DataModule`.
        num_workers: The ``num_workers`` argument to pass to the :class:`~flash.core.
↪data.data_module.DataModule`.
        preprocess_kwargs: Additional keyword arguments to use when constructing the␣
↪preprocess. Will only be used
            if ``preprocess = None``.

    Returns:
        The constructed data module.
    """
    return super().from_data_source(
        "sklearn",
        train_bunch,
        val_bunch,
        test_bunch,
        predict_bunch,
        train_transform=train_transform,
        val_transform=val_transform,
        test_transform=test_transform,
        predict_transform=predict_transform,
        data_fetcher=data_fetcher,
        preprocess=preprocess,
        val_split=val_split,
        batch_size=batch_size,
        num_workers=num_workers,
        **preprocess_kwargs,
    )
```

The final step is to implement the `num_features` property for our `TemplateData`. This is just a convenience for the user that finds the `num_features` attribute on any of the data sets and returns it. Here's the code:

```
@property
def num_features(self) -> Optional[int]:
    """Tries to get the ``num_features`` from each dataset in turn and returns the␣
↪output."""
    return (
        getattr(self.train_dataset, "num_features", None) or getattr(self.val_dataset,
↪ "num_features", None)
        or getattr(self.test_dataset, "num_features", None)
    )
```

25.4 BaseVisualization

An optional step is to implement a *BaseVisualization*. The *BaseVisualization* lets you control how data at various points in the pipeline can be visualized. This is extremely useful for debugging purposes, allowing users to view their data and understand the impact of their transforms.

Note: Don't worry about implementing it right away, you can always come back and add it later!

Here's the code for our `TemplateVisualization` which just prints the data:

```
class TemplateVisualization(BaseVisualization):
    """The ``TemplateVisualization`` class is a :class:`~flash.core.data.callbacks.
↪BaseVisualization` that just prints
    the data. If you want to provide a visualization with your task, you can override␣
↪these hooks."""

    def show_load_sample(self, samples: List[Any], running_stage: RunningStage):
        print(samples)

    def show_pre_tensor_transform(self, samples: List[Any], running_stage:␣
↪RunningStage):
        print(samples)

    def show_to_tensor_transform(self, samples: List[Any], running_stage:␣
↪RunningStage):
        print(samples)

    def show_post_tensor_transform(self, samples: List[Any], running_stage:␣
↪RunningStage):
        print(samples)

    def show_per_batch_transform(self, batch: List[Any], running_stage):
        print(batch)
```

We can configure our custom visualization in the `TemplateData` using *configure_data_fetcher()* like this:

```
@staticmethod
def configure_data_fetcher(*args, **kwargs) -> BaseDataFetcher:
    """We can, *optionally*, provide a data visualization callback using the␣
↪``configure_data_fetcher`` method."""
    return TemplateVisualization(*args, **kwargs)
```

25.5 Postprocess

Postprocess contains any transforms that need to be applied *after* the model. You may want to use it for: converting tokens back into text, applying an inverse normalization to an output image, resizing a generated image back to the size of the input, etc. As an example, here's the `TextClassificationPostprocess` which gets the logits from a `SequenceClassifierOutput`:

```
class TextClassificationPostprocess(Postprocess):
```

(continues on next page)

(continued from previous page)

```
    def per_batch_transform(self, batch: Any) -> Any:
        if isinstance(batch, SequenceClassifierOutput):
            batch = batch.logits
        return super().per_batch_transform(batch)
```

In your *DataSource* or *Preprocess*, you can add metadata to the batch using the *METADATA* key. Your *Postprocess* can then use this metadata in its transforms. You should use this approach if your postprocessing depends on the state of the input before the *Preprocess* transforms. For example, if you want to resize the predictions to the original size of the inputs you should add the original image size in the *METADATA*. Here's an example from the `SemanticSegmentationNumpyDataSource`:

```
def load_sample(self, sample: Dict[str, Any], dataset: Optional[Any] = None) ->␣
↪Dict[str, Any]:
    img = torch.from_numpy(sample[DefaultDataKeys.INPUT]).float()
    sample[DefaultDataKeys.INPUT] = img
    sample[DefaultDataKeys.METADATA] = img.shape
    return sample
```

The *METADATA* can now be referenced in your *Postprocess*. For example, here's the code for the `per_sample_transform` method of the `SemanticSegmentationPostprocess`:

```
def per_sample_transform(self, sample: Any) -> Any:
    resize = K.geometry.Resize(sample[DefaultDataKeys.METADATA][-2:], interpolation=
↪'bilinear')
    sample[DefaultDataKeys.PREDS] = resize(torch.stack(sample[DefaultDataKeys.PREDS]))
    sample[DefaultDataKeys.INPUT] = resize(torch.stack(sample[DefaultDataKeys.INPUT]))
    return super().per_sample_transform(sample)
```

Now that you've got some data, it's time to *add some backbones for your task!*

CHAPTER

TWENTYSIX

THE BACKBONES

Now that you've got a way of loading data, you should implement some backbones to use with your *Task*. Create a *FlashRegistry* to use with your *Task* in backbones.py.

The registry allows you to register backbones for your task that can be selected by the user. The backbones can come from anywhere as long as you can register a function that loads the backbone. Furthermore, the user can add their own models to the existing backbones, without having to write their own *Task*!

You can create a registry like this:

```
TEMPLATE_BACKBONES = FlashRegistry("backbones")
```

Let's add a simple MLP backbone to our registry. We need a function that creates the backbone and returns it along with the output size (so that we can create the model head in our *Task*). You can use any name for the function, although we use `load_{model name}` by convention. Here's the code:

```
@TEMPLATE_BACKBONES(name="mlp-128", namespace="template/classification")
def load_mlp_128(num_features, **_):
    """A simple MLP backbone with 128 hidden units."""
    return nn.Sequential(
        nn.Linear(num_features, 128),
        nn.ReLU(True),
        nn.BatchNorm1d(128),
    ), 128
```

Here's another example with a slightly more complex model:

```
@TEMPLATE_BACKBONES(name="mlp-128-256", namespace="template/classification")
def load_mlp_128_256(num_features, **_):
    """An two layer MLP backbone with 128 and 256 hidden units respectively."""
    return nn.Sequential(
        nn.Linear(num_features, 128),
        nn.ReLU(True),
        nn.BatchNorm1d(128),
        nn.Linear(128, 256),
        nn.ReLU(True),
        nn.BatchNorm1d(256),
    ), 256
```

Here's a more advanced example, which adds `SimCLR` to the `IMAGE_CLASSIFIER_BACKBONES`, from flash/image/backbones.py:

```
@IMAGE_CLASSIFIER_BACKBONES(name="simclr-imagenet", namespace="vision", package="bolts
↪")
def load_simclr_imagenet(path_or_url: str = f"{ROOT_S3_BUCKET}/simclr/bolts_simclr_
↪imagenet/simclr_imagenet.ckpt", **_):
```

(continues on next page)

(continued from previous page)

```
    simclr: LightningModule = SimCLR.load_from_checkpoint(path_or_url, strict=False)
    # remove the last two layers & turn it into a Sequential model
    backbone = nn.Sequential(*list(simclr.encoder.children())[:-2])
    return backbone, 2048
```

Once you've got some data and some backbones, *implement your task!*

CHAPTER

TWENTYSEVEN

THE TASK

Once you've implemented a Flash *DataModule* and some backbones, you should implement your *Task* in model.py. The *Task* is responsible for: setting up the backbone, performing the forward pass of the model, and calculating the loss and any metrics. Remember that, under the hood, the Flash *Task* is simply a `LightningModule` with some helpful defaults.

To build your task, you can start by overriding the base *Task* or any of the existing *Task* implementations. For example, in our scikit-learn example, we can just override `ClassificationTask` which provides good defaults for classification.

You should attach your backbones registry as a class attribute like this:

```
class TemplateSKLearnClassifier(ClassificationTask):

    backbones: FlashRegistry = TEMPLATE_BACKBONES
```

27.1 Model architecture and hyper-parameters

In the `__init__()`, you will need to configure defaults for the:

- loss function
- optimizer
- metrics
- backbone / model

You will also need to create the backbone from the registry and create the model head. Here's the code:

```
def __init__(
    self,
    num_features: int,
    num_classes: int,
    backbone: Union[str, Tuple[nn.Module, int]] = "mlp-128",
    backbone_kwargs: Optional[Dict] = None,
    loss_fn: Optional[Callable] = None,
    optimizer: Union[Type[torch.optim.Optimizer], torch.optim.Optimizer] = torch.
↪optim.Adam,
    optimizer_kwargs: Optional[Dict[str, Any]] = None,
    scheduler: Optional[Union[Type[_LRScheduler], str, _LRScheduler]] = None,
    scheduler_kwargs: Optional[Dict[str, Any]] = None,
    metrics: Union[torchmetrics.Metric, Mapping, Sequence, None] = None,
    learning_rate: float = 1e-2,
```

(continues on next page)

(continued from previous page)

```
    multi_label: bool = False,
    serializer: Optional[Union[Serializer, Mapping[str, Serializer]]] = None,
):
    super().__init__(
        model=None,
        loss_fn=loss_fn,
        optimizer=optimizer,
        optimizer_kwargs=optimizer_kwargs,
        scheduler=scheduler,
        scheduler_kwargs=scheduler_kwargs,
        metrics=metrics,
        learning_rate=learning_rate,
        multi_label=multi_label,
        serializer=serializer,
    )

    self.save_hyperparameters()

    if not backbone_kwargs:
        backbone_kwargs = {}

    if isinstance(backbone, tuple):
        self.backbone, out_features = backbone
    else:
        self.backbone, out_features = self.backbones.get(backbone)(num_features=num_
↪features, **backbone_kwargs)

    self.head = nn.Linear(out_features, num_classes)
```

Note: We call `save_hyperparameters()` to log the arguments to the `__init__` as hyperparameters. Read more here.

27.2 Adding the model routines

You should override the `{train,val,test,predict}_step` methods. The default `{train,val,test,predict}_step` implementations in *`Task`* expect a tuple containing the input (to be passed to the model) and target (to be used when computing the loss), and should be suitable for most applications. In our template example, we just extract the input and target from the input mapping and forward them to the `super` methods. Here's the code for the `training_step`:

```
def training_step(self, batch: Any, batch_idx: int) -> Any:
    """For the training step, we just extract the :attr:`~flash.core.data.data_source.
↪DefaultDataKeys.INPUT` and
    :attr:`~flash.core.data.data_source.DefaultDataKeys.TARGET` keys from the input␣
↪and forward them to the
    :meth:`~flash.core.model.Task.training_step`."""
    batch = (batch[DefaultDataKeys.INPUT], batch[DefaultDataKeys.TARGET])
    return super().training_step(batch, batch_idx)
```

We use the same code for the `validation_step` and `test_step`. For `predict_step` we don't need the targets, so our code looks like this:

```
def predict_step(self, batch: Any, batch_idx: int, dataloader_idx: int = 0) -> Any:
    """For the predict step, we just extract the :attr:`~flash.core.data.data_source.
↪DefaultDataKeys.INPUT` key from
    the input and forward it to the :meth:`~flash.core.model.Task.predict_step`."""
    batch = (batch[DefaultDataKeys.INPUT])
    return super().predict_step(batch, batch_idx, dataloader_idx=dataloader_idx)
```

Note: You can completely replace the `{train,val,test,predict}_step` methods (that is, without a call to `super`) if you need more custom behaviour for your *`Task`* at a particular stage.

Finally, we use our backbone and head in a custom forward pass:

```
def forward(self, x) -> torch.Tensor:
    """First call the backbone, then the model head."""
    x = self.backbone(x)
    return self.head(x)
```

Now that you've got your task, take a look at some *optional advanced features you can add* or go ahead and *create some examples showing your task in action!*

CHAPTER

TWENTYEIGHT

OPTIONAL EXTRAS

28.1 Organize your transforms in transforms.py

If you have a lot of default transforms, it can be useful to put them all in a `transforms.py` file, to be referenced in your *Preprocess*. Here's an example from image/classification/transforms.py which creates some default transforms given the desired image size:

```
def default_transforms(image_size: Tuple[int, int]) -> Dict[str, Callable]:
    """The default transforms for image classification: resize the image, convert the␣
↪image and target to a tensor,
    collate the batch, and apply normalization."""
    if _KORNIA_AVAILABLE and not os.getenv("FLASH_TESTING", "0") == "1":
        #  Better approach as all transforms are applied on tensor directly
        return {
            "to_tensor_transform": nn.Sequential(
                ApplyToKeys(DefaultDataKeys.INPUT, torchvision.transforms.ToTensor()),
                ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
            ),
            "post_tensor_transform": ApplyToKeys(
                DefaultDataKeys.INPUT,
                K.geometry.Resize(image_size),
            ),
            "collate": kornia_collate,
            "per_batch_transform_on_device": ApplyToKeys(
                DefaultDataKeys.INPUT,
                K.augmentation.Normalize(torch.tensor([0.485, 0.456, 0.406]), torch.
↪tensor([0.229, 0.224, 0.225])),
            )
        }
    else:
        return {
            "pre_tensor_transform": ApplyToKeys(DefaultDataKeys.INPUT, T.Resize(image_
↪size)),
            "to_tensor_transform": nn.Sequential(
                ApplyToKeys(DefaultDataKeys.INPUT, torchvision.transforms.ToTensor()),
                ApplyToKeys(DefaultDataKeys.TARGET, torch.as_tensor),
            ),
            "post_tensor_transform": ApplyToKeys(
                DefaultDataKeys.INPUT,
                T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
            ),
            "collate": kornia_collate,
        }
```

Here's how we create our transforms in the `ImageClassificationPreprocess`:

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

28.2 Add output serializers to your Task

We recommend that you do most of the heavy lifting in the *`Postprocess`*. Specifically, it should include any formatting and transforms that should always be applied to the predictions. If you want to support different use cases that require different prediction formats, you should add some *`Serializer`* implementations in a `serialization.py` file.

Some good examples are in flash/core/classification.py. Here's the *`Classes Serializer`*:

```
class Classes(ClassificationSerializer):
    """A :class:`.Serializer` which applies an argmax to the model outputs (either␣
↪logits or probabilities) and
    converts to a list.

    Args:
        multi_label: If true, treats outputs as multi label logits.

        threshold: The threshold to use for multi_label classification.
    """

    def __init__(self, multi_label: bool = False, threshold: float = 0.5):
        super().__init__(multi_label)

        self.threshold = threshold

    def serialize(self, sample: Any) -> Union[int, List[int]]:
        if self.multi_label:
            one_hot = (sample.sigmoid() > self.threshold).int().tolist()
            result = []
            for index, value in enumerate(one_hot):
                if value == 1:
                    result.append(index)
            return result
        return torch.argmax(sample, -1).tolist()
```

Alternatively, here's the *`Logits Serializer`*:

```
class Logits(ClassificationSerializer):
    """A :class:`.Serializer` which simply converts the model outputs (assumed to be␣
↪logits) to a list."""

    def serialize(self, sample: Any) -> Any:
        return sample.tolist()
```

Take a look at *Predictions (inference)* to learn more.

Once you've added any optional extras, it's time to *create some examples showing your task in action!*

CHAPTER

TWENTYNINE

THE EXAMPLES

Now you've implemented your task, it's time to add some examples showing how cool it is! We usually provide one finetuning example in flash_examples/finetuning and one predict / inference example in flash_examples/predict. You can base these off of our `template.py` examples. Let's take a closer look.

29.1 finetuning

The finetuning example should:

1. download the data (we'll add the example to our CI later on, so choose a dataset small enough that it runs in reasonable time)
2. load the data into a *DataModule*
3. create an instance of the *Task*
4. create a *Trainer*
5. call *finetune()* or *fit()* to train your model
6. save the checkpoint
7. generate predictions for a few examples *(optional)*

For our template example we don't have a pretrained backbone, so we can just call *fit()* rather than *finetune()*. Here's the full example (flash_examples/finetuning/template.py):

```
import numpy as np
from sklearn import datasets

import flash
from flash.core.classification import Labels
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Download the data
data_bunch = datasets.load_iris()

# 2. Load the data
datamodule = TemplateData.from_sklearn(
    train_bunch=data_bunch,
    val_split=0.8,
)

# 3. Build the model
model = TemplateSKLearnClassifier(
```

(continues on next page)

(continued from previous page)

```
    num_features=datamodule.num_features,
    num_classes=datamodule.num_classes,
    serializer=Labels(),
)

# 4. Create the trainer.
trainer = flash.Trainer(max_epochs=1, limit_train_batches=1, limit_val_batches=1)

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

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

# 7. Classify a few examples
predictions = model.predict([
    np.array([4.9, 3.0, 1.4, 0.2]),
    np.array([6.9, 3.2, 5.7, 2.3]),
    np.array([7.2, 3.0, 5.8, 1.6]),
])
print(predictions)
```

We get this output:

```
['setosa', 'virginica', 'versicolor']
```

29.2 predict

The predict example should:

1. download the data (this should be the data from the finetuning example)
2. load an instance of the *Task* from a checkpoint stored on *S3* (speak with one of us about getting your checkpoint hosted)
3. generate predictions for a few examples
4. generate predictions for a whole dataset, folder, etc.

For our template example we don't have a pretrained backbone, so we can just call *fit()* rather than *finetune()*. Here's the full example (flash_examples/predict/template.py):

```
import numpy as np
from sklearn import datasets

from flash import Trainer
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Download the data
data_bunch = datasets.load_iris()

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

# 3. Classify a few examples
```

(continues on next page)

(continued from previous page)

```
predictions = model.predict([
    np.array([4.9, 3.0, 1.4, 0.2]),
    np.array([6.9, 3.2, 5.7, 2.3]),
    np.array([7.2, 3.0, 5.8, 1.6]),
])
print(predictions)

# 4. Or generate predictions from a whole dataset!
datamodule = TemplateData.from_sklearn(predict_bunch=data_bunch)

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

We get this output:

```
['setosa', 'virginica', 'versicolor']
[['setosa', 'setosa', 'setosa', 'setosa'], ..., ['virginica', 'virginica']]
```

Now that you've got some examples showing your awesome task in action, it's time to *write some tests!*

CHAPTER

THIRTY

THE TESTS

Our next step is to create some tests for our *Task*. For the `TemplateSKLearnClassifier`, we will just create some basic tests. You should expand on these to include tests for any specific functionality you have in your *Task*.

30.1 Smoke tests

We use smoke tests, usually called `test_smoke`, throughout. These just instantiate the class we are testing, to see that they can be created without raising any errors.

30.2 tests/examples/test_scripts.py

Before we write our custom tests, we should add out examples to the CI. To do this, add a line for each example (`finetuning` and `predict`) to the annotation of `test_example` in tests/examples/test_scripts.py. Here's how those lines look for our `template.py` examples:

```
pytest.param(
    "finetuning",
    "template.py",
    marks=pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
),
...
pytest.param(
    "predict",
    "template.py",
    marks=pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
),
```

30.3 test_data.py

The most important tests in test_data.py check that the `from_*` methods work correctly. In the class `TestTemplateData`, we have two of these: `test_from_numpy` and `test_from_sklearn`. In general, there should be one `test_from_*` method for each `data_source` you have configured.

Here's the code for `test_from_numpy`:

```
    def test_from_numpy(self):
        """Tests that ``TemplateData`` is properly created when using the ``from_
↪numpy`` method."""
```

(continues on next page)

(continued from previous page)

```
        data = np.random.rand(10, self.num_features)
        targets = np.random.randint(0, self.num_classes, (10, ))

        # instantiate the data module
        dm = TemplateData.from_numpy(
            train_data=data,
            train_targets=targets,
            val_data=data,
            val_targets=targets,
            test_data=data,
            test_targets=targets,
            batch_size=2,
            num_workers=0,
        )
        assert dm is not None
        assert dm.train_dataloader() is not None
        assert dm.val_dataloader() is not None
        assert dm.test_dataloader() is not None

        # check training data
        data = next(iter(dm.train_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2, )

        # check val data
        data = next(iter(dm.val_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2, )

        # check test data
        data = next(iter(dm.test_dataloader()))
        rows, targets = data[DefaultDataKeys.INPUT], data[DefaultDataKeys.TARGET]
        assert rows.shape == (2, self.num_features)
        assert targets.shape == (2, )
```

30.4 test_model.py

In test_model.py, we first have `test_forward` and `test_train`. These test that tensors can be passed to the forward and that the *`Task`* can be trained. Here's the code for `test_forward` and `test_train`:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
@pytest.mark.parametrize("num_classes", [4, 256])
@pytest.mark.parametrize("shape", [(1, 3), (2, 128)])
def test_forward(num_classes, shape):
    """Tests that a tensor can be given to the model forward and gives the correct␣
↪output size."""
    model = TemplateSKLearnClassifier(
        num_features=shape[1],
        num_classes=num_classes,
    )
    model.eval()
```

(continues on next page)

(continued from previous page)

```
    row = torch.rand(*shape)

    out = model(row)
    assert out.shape == (shape[0], num_classes)
```

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_train(tmpdir):
    """Tests that the model can be trained on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    train_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.fit(model, train_dl)
```

We also include tests for validating and testing: `test_val`, and `test_test`. These tests are very similar to `test_train`, but here they are for completeness:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_val(tmpdir):
    """Tests that the model can be validated on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    val_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.validate(model, val_dl)
```

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_test(tmpdir):
    """Tests that the model can be tested on our ``DummyDataset``."""
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    test_dl = torch.utils.data.DataLoader(DummyDataset(), batch_size=4)
    trainer = Trainer(default_root_dir=tmpdir, fast_dev_run=True)
    trainer.test(model, test_dl)
```

We also include tests for prediction named `test_predict_*` for each of our data sources. In our case, we have `test_predict_numpy` and `test_predict_sklearn`. These tests should use the `data_source` argument to *`predict()`* to select the required `DataSource`. Here's `test_predict_sklearn` as an example:

```
@pytest.mark.skipif(not _SKLEARN_AVAILABLE, reason="sklearn isn't installed")
def test_predict_sklearn():
    """Tests that we can generate predictions from a scikit-learn ``Bunch``."""
    bunch = datasets.load_iris()
    model = TemplateSKLearnClassifier(num_features=DummyDataset.num_features, num_
↪classes=DummyDataset.num_classes)
    data_pipe = DataPipeline(preprocess=TemplatePreprocess())
```

(continues on next page)

(continued from previous page)

```
    out = model.predict(bunch, data_source="sklearn", data_pipeline=data_pipe)
    assert isinstance(out[0], int)
```

Now that you've written the tests, it's time to *add some docs!*

CHAPTER

THIRTYONE

THE DOCS

The final step is to add some docs. For each *Task* in Flash, we have a docs page in docs/source/reference. You should create a `.rst` file there with the following:

- a brief description of the task
- the predict example
- the finetuning example
- any relevant API reference

Here are the contents of docs/source/reference/template.rst which breaks down each of these steps:

```
.. _template:

########
Template
########

********
The task
********

Here you should add a description of your task. For example:
Classification is the task of assigning one of a number of classes to each data point.
The :class:`~flash.template.TemplateSKLearnClassifier` is a :class:`~flash.core.model.
↪Task` for classifying the datasets included with scikit-learn.

------

*********
Inference
*********

Here, you should add a short intro to your predict example, and then use␣
↪``literalinclude`` to add it.

.. note:: We skip the first 14 lines as they are just the copyright notice.

Our predict example uses a model pre-trained on the Iris data.

.. literalinclude:: ../../../flash_examples/predict/template.py
    :language: python
    :lines: 14-
```

(continues on next page)

(continued from previous page)

```
For more advanced inference options, see :ref:`predictions`.

------

********
Training
********

In this section, we briefly describe the data, and then ``literalinclude`` our␣
↪finetuning example.

Now we'll train on Fisher's classic Iris data.
It contains 150 records with four features (sepal length, sepal width, petal length,␣
↪and petal width) in three classes (species of Iris: setosa, virginica and␣
↪versicolor).

Now all we need is to train our task!

.. literalinclude:: ../../../flash_examples/finetuning/template.py
    :language: python
    :lines: 14-

------

*************
API reference
*************

We usually include the API reference for the :class:`~flash.core.model.Task` and␣
↪:class:`~flash.core.data.data_module.DataModule`.
You can optionally add the other classes you've implemented.
To add the API reference, use the ``autoclass`` directive.

.. _template_classifier:

TemplateSKLearnClassifier
-------------------------

.. autoclass:: flash.template.TemplateSKLearnClassifier
    :members:
    :exclude-members: forward

.. _template_data:

TemplateData
------------

.. autoclass:: flash.template.TemplateData
```

Here's the rendered doc page!

Once the docs are done, it's finally time to open a PR and wait for some reviews!

Congratulations on adding your first `Task` to Flash, we hope to see you again soon!

CHAPTER

THIRTYTWO

TEMPLATE

32.1 The task

Here you should add a description of your task. For example: Classification is the task of assigning one of a number of classes to each data point. The *TemplateSKLearnClassifier* is a *Task* for classifying the datasets included with scikit-learn.

32.2 Inference

Here, you should add a short intro to your predict example, and then use `literalinclude` to add it.

Note: We skip the first 14 lines as they are just the copyright notice.

Our predict example uses a model pre-trained on the Iris data.

```
import numpy as np
from sklearn import datasets

from flash import Trainer
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Download the data
data_bunch = datasets.load_iris()

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

# 3. Classify a few examples
predictions = model.predict([
    np.array([4.9, 3.0, 1.4, 0.2]),
    np.array([6.9, 3.2, 5.7, 2.3]),
    np.array([7.2, 3.0, 5.8, 1.6]),
])
print(predictions)

# 4. Or generate predictions from a whole dataset!
datamodule = TemplateData.from_sklearn(predict_bunch=data_bunch)
```

(continues on next page)

(continued from previous page)

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

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

32.3 Training

In this section, we briefly describe the data, and then `literalinclude` our finetuning example.

Now we'll train on Fisher's classic Iris data. It contains 150 records with four features (sepal length, sepal width, petal length, and petal width) in three classes (species of Iris: setosa, virginica and versicolor).

Now all we need is to train our task!

```
import numpy as np
from sklearn import datasets

import flash
from flash.core.classification import Labels
from flash.template import TemplateData, TemplateSKLearnClassifier

# 1. Download the data
data_bunch = datasets.load_iris()

# 2. Load the data
datamodule = TemplateData.from_sklearn(
    train_bunch=data_bunch,
    val_split=0.8,
)

# 3. Build the model
model = TemplateSKLearnClassifier(
    num_features=datamodule.num_features,
    num_classes=datamodule.num_classes,
    serializer=Labels(),
)

# 4. Create the trainer.
trainer = flash.Trainer(max_epochs=1, limit_train_batches=1, limit_val_batches=1)

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

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

# 7. Classify a few examples
predictions = model.predict([
    np.array([4.9, 3.0, 1.4, 0.2]),
    np.array([6.9, 3.2, 5.7, 2.3]),
    np.array([7.2, 3.0, 5.8, 1.6]),
])
print(predictions)
```

32.4 API reference

We usually include the API reference for the *Task* and *DataModule*. You can optionally add the other classes you've implemented. To add the API reference, use the `autoclass` directive.

32.4.1 TemplateSKLearnClassifier

class flash.template.**TemplateSKLearnClassifier**(*num_features*, *num_classes*, *backbone='mlp-128'*, *backbone_kwargs=None*, *loss_fn=None*, *optimizer=torch.optim.Adam*, *optimizer_kwargs=None*, *scheduler=None*, *scheduler_kwargs=None*, *metrics=None*, *learning_rate=0.01*, *multi_label=False*, *serializer=None*)

The `TemplateSKLearnClassifier` is a `ClassificationTask` that classifies tabular data from scikit-learn.

Parameters

- **num_features** (`int`) – The number of features (elements) in the input data.
- **num_classes** (`int`) – The number of classes (outputs) for this *Task*.
- **backbone** (`Union[str, Tuple[Module, int]]`) – The backbone name (or a tuple of `nn.Module`, output size) to use.
- **backbone_kwargs** (`Optional[Dict]`) – Any additional kwargs to pass to the backbone constructor.
- **loss_fn** (`Optional[Callable]`) – The loss function to use. If `None`, a default will be selected by the `ClassificationTask` depending on the `multi_label` argument.
- **optimizer** (`Union[Type[Optimizer], Optimizer]`) – The optimizer or optimizer class to use.
- **optimizer_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the optimizer (if not passed as an instance).
- **scheduler** (`Union[Type[LRScheduler], str, LRScheduler, None]`) – The scheduler or scheduler class to use.
- **scheduler_kwargs** (`Optional[Dict[str, Any]]`) – Additional kwargs to use when creating the scheduler (if not passed as an instance).
- **metrics** (`Union[Metric, Mapping, Sequence, None]`) – Any metrics to use with this *Task*. If `None`, a default will be selected by the `ClassificationTask` depending on the `multi_label` argument.
- **learning_rate** (`float`) – The learning rate for the optimizer.
- **multi_label** (`bool`) – If `True`, this will be treated as a multi-label classification problem.
- **serializer** (`Union[Serializer, Mapping[str, Serializer], None]`) – The *Serializer* to use for prediction outputs.

predict_step(*batch*, *batch_idx*, *dataloader_idx=0*)
For the predict step, we just extract the *INPUT* key from the input and forward it to the `predict_step()`.

Return type Any

test_step(*batch*, *batch_idx*)
For the test step, we just extract the *INPUT* and *TARGET* keys from the input and forward them to the `test_step()`.

Return type Any

training_step(*batch*, *batch_idx*)
For the training step, we just extract the *INPUT* and *TARGET* keys from the input and forward them to the `training_step()`.

Return type Any

validation_step(*batch*, *batch_idx*)
For the validation step, we just extract the *INPUT* and *TARGET* keys from the input and forward them to the `validation_step()`.

Return type Any

32.4.2 TemplateData

class flash.template.**TemplateData**(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *predict_dataset=None*, *data_source=None*, *preprocess=None*, *postprocess=None*, *data_fetcher=None*, *val_split=None*, *batch_size=1*, *num_workers=None*)
Creating our *DataModule* is as easy as setting the `preprocess_cls` attribute. We get the `from_numpy` method for free as we've configured a `DefaultDataSources.NUMPY` data source. We'll also add a `from_sklearn` method so that we can use our `TemplateSKLearnDataSource`. Finally, we define the ``num_features` property for convenience.

CHAPTER

THIRTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

G

I

J

L

M

N

O

P

S

T

U

V