Deep Learning Workshop: Applied Computer Vision

Jeremy Pinto, jeremy.pinto@mila.quebec

Pierre-Luc St-Charles, pierreluc.stcharles@mila.quebec

Session 4

Course Structure

The course is spread over **4 sessions**, with a mix of theory and practical content.

Session 1: Theory	Session 2: Practical	Session 3: Theory	Session 4: Practical
Introduction to computer vision and deep learning concepts	**Implementation of a classification algorithm on a rock, paper, scissors dataset**	**Review of state-of-the-art models, vision transformers, object detection**	**Overview of an applied project using modern tools and libraries and best practices**

Introduction

Introduction

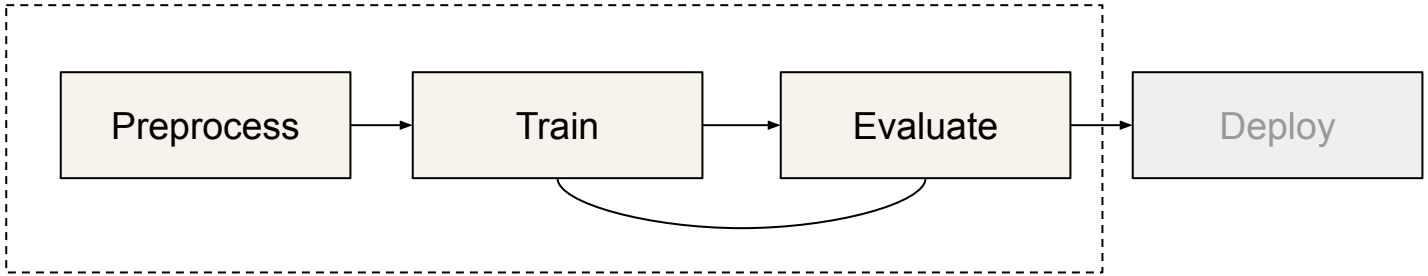
In this presentation, we will look at an **applied deep learning project** implemented on a big computer-vision dataset.

We will be giving an overview of the **various components** that go into an applied project with best practices in mind.

Photo by Todd Quackenbush on Unsplash

Focus

We will be focusing on the data preprocessing, training and evaluation pipelines. Deployment is not considered in this session.



Note

There are many different **design decisions** that go into building a deep learning project.

Some tools might be better suited than others in **different circumstances**.

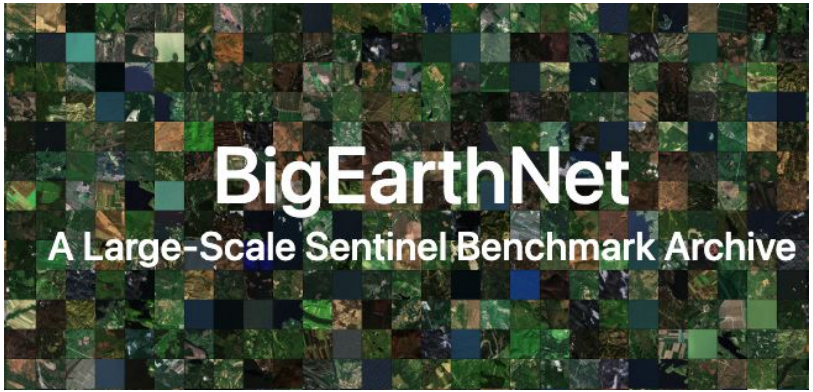
Fuji will be reminding us from time to time about **alternative** tools he might have used.

BigEarthNet

BigEarthNet

The dataset we will be using is the BigEarthNet **remote-sensing** dataset, consisting of hundreds of thousands of **satellite images**.

The goal of the dataset is to **classify** satellite images regions based on their content.

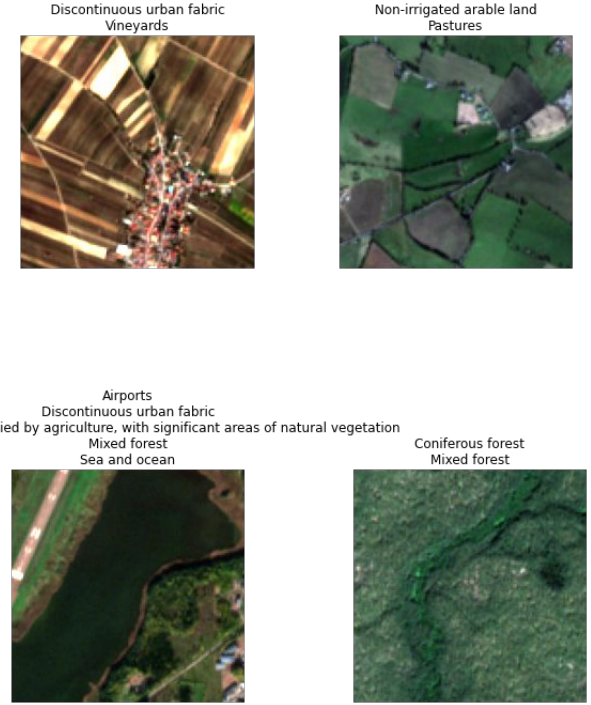


BigEarthNet - Stats

A total of **590 000** images were collected.

43 land-cover class labels were attributed to each image using the 2018 CORINE Land Cover database.

Each image can have **multiple labels**. 95% of images have at most 5 labels.



[dataset paper]

Data Distribution

Here is the full list of **43 classes** and their respective counts in the BigEarthNet dataset.

Notice the data is **not balanced** and certain classes dominate over others (e.g. forests vs. airports).

Table 2: The considered Level-3 CLC classes and the number of images associated with each land-cover class in the BigEarthNet.

Land-Cover Classes	Number of Images
Mixed forest	217, 119
Coniferous forest	211, 703
Non-irrigated arable land	196, 695
Transitional woodland/shrub	173, 506
Broad-leaved forest	150, 944
Land principally occupied by agriculture, with significant areas of natural vegetation	147, 095
Complex cultivation patterns	107, 786
Pastures	103, 554
Water bodies	83, 811
Sea and ocean	81, 612
Discontinuous urban fabric	69, 872
Agro-forestry areas	30, 674
Peatbogs	23, 207
Permanently irrigated land	13589
Industrial or commercial units	12895
Natural grassland	12, 835
Olive groves	12, 538
Sclerophyllous vegetation	11, 241
Continuous urban fabric	10, 784
Water courses	10, 572
Vineyards	9, 567
Annual crops associated with permanent crops	7, 022
Inland marshes	6, 236
Moors and heathland	5, 890
Sport and leisure facilities	5, 353
Fruit trees and berry plantations	4, 754
Mineral extraction sites	4, 618
Rice fields	3, 793
Road and rail networks and associated land	3, 384
Bare rock	3, 277
Green urban areas	1, 786
Beaches, dunes, sands	1, 578
Sparsely vegetated areas	1, 563
Salt marshes	1, 562
Coastal lagoons	1, 498
Construction sites	1, 174
Estuaries	1, 086
Intertidal flats	1, 003
Airports	979
Dump sites	959
Port areas	509
Salines	424
Burnt areas	328

Source

Sentinel-2

All images were acquired using the **Sentinel-2 satellite**.

Images were taken over **10 european countries** between **June 2017 and May 2018**.

Sentinel-2 satellite

source

Spectral Bands

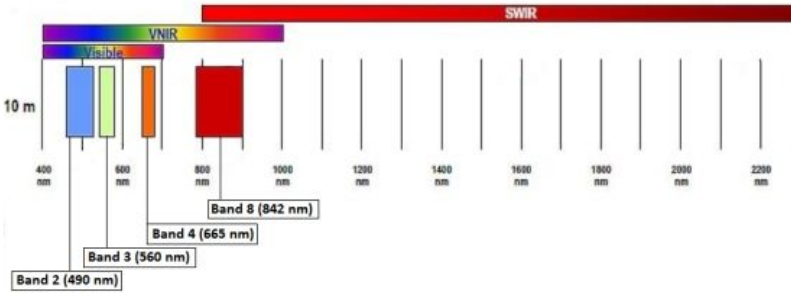
Sentinel-2 records images across 13 different **spectral bands** (channels).

We will only be considering three bands:

Band 2: ~Blue

Band 3: ~Green

Band 4: ~Red



source

Loss Function

Since we can have multiple labels per image, we will treat each label as **independent** and make a binary prediction for each class.

We can conveniently use the BCEWithLogitsLoss() function to do this in pytorch.

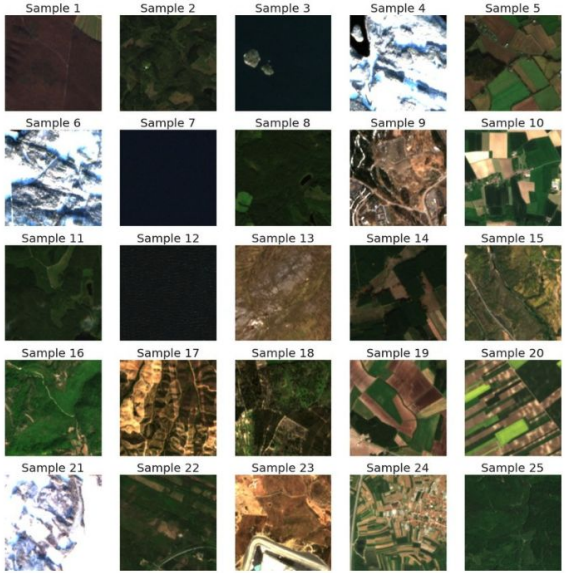
```python
import torch
def to_tensor(x):
    """Convert list to tensor."""
    return torch.tensor(x, dtype=torch.float)

loss_fn = torch.nn.BCEWithLogitsLoss()

outputs = [[100, -100, 100,  100, -100]]  # Outputs before sigmoid
targets = [[  1,    0,   1,    1,    0]]  # 1 sample, 5 classes

loss = loss_fn(to_tensor(outputs), to_tensor(targets))
print(f"Loss: {loss}") # Expected ~0 loss
```

```
Loss: 0.0
```

Data Splits

We will be using the **same splits** used in the dataset's recommended tools.

These splits **remove** images covered in **snow/clouds** as show in this figure.

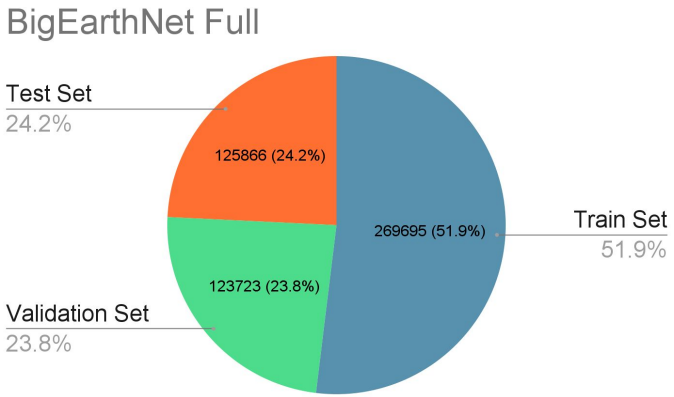


source

Data Splits

As part of this project, we have prepared and released **3 separate subsets** of the BigEarthNet dataset (Note that splits were preserved).

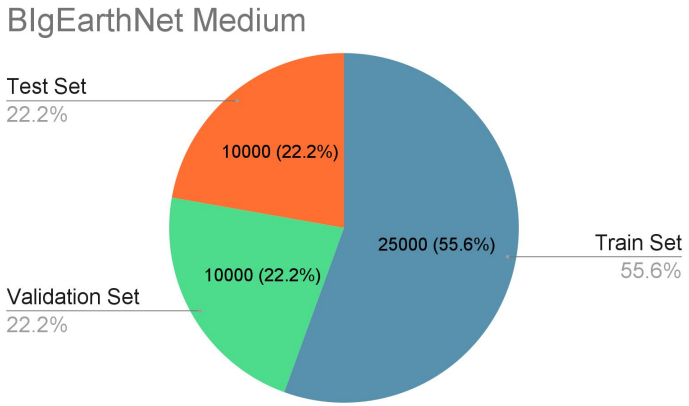
- **BigEarthNet-full** The full dataset using the same splits as in the suggested tools. It will be used to train **larger models**.



Data Splits

As part of this project, we have prepared and released **3 separate subsets** of the BigEarthNet dataset (Note that splits were preserved).

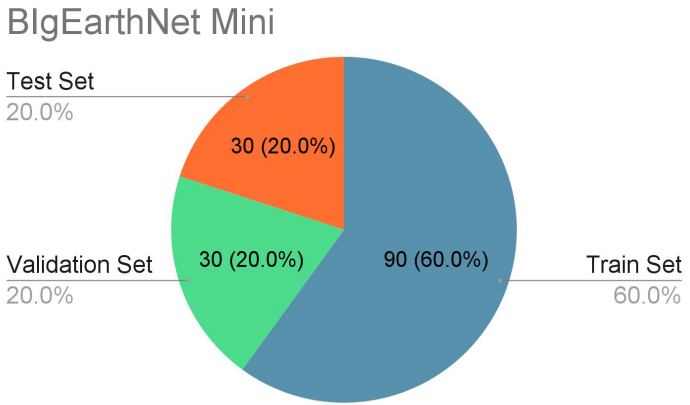
- **BigEarthNet-medium** ~10% of the full dataset. It will be used to guide hyper-parameter searches and model selection.



Data Splits

As part of this project, we have prepared and released **3 separate subsets** of the BigEarthNet dataset (Note that splits were preserved).

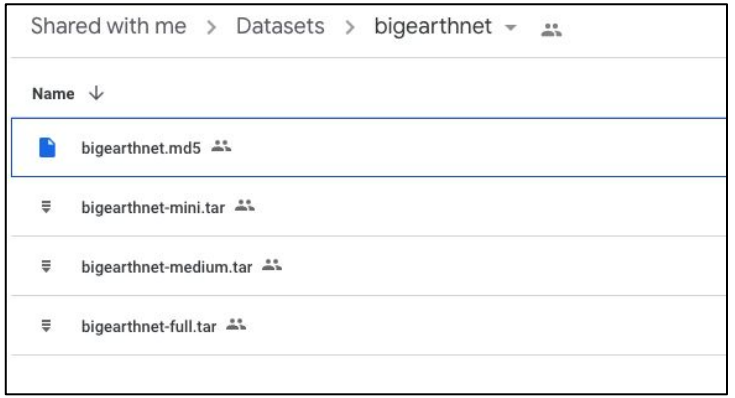
- **BigEarthNet-mini** A tiny subset of the original dataset (150 samples). It is meant to be used for **testing the code**, **running locally** and in CI/CD pipelines.



Download datasets

The datasets are made available via a shared google drive. You **do not need** to download the files manually.

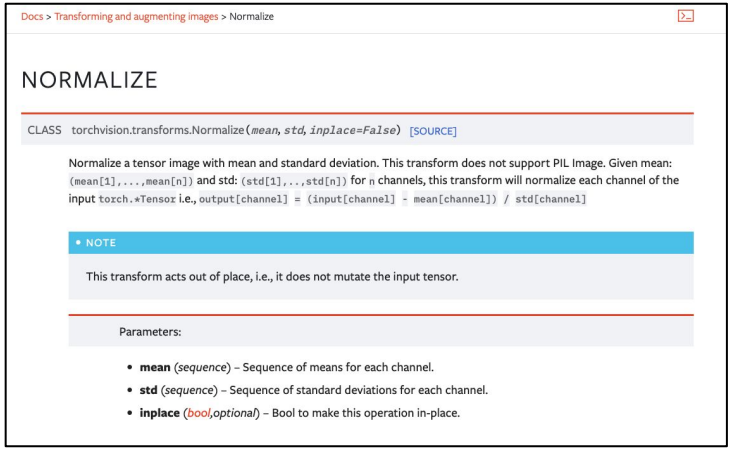
The code will automatically download and extract them when specified.



Normalization

One thing to note about these spectral bands is that **they are not** in a standard pixel range (0-255).

To sanitize the inputs to our neural network, we will be computing the **mean** and **standard deviation** of pixels **channel-wise**, and normalizing each channel according to the train set statistics. Stats are computed on the train set.



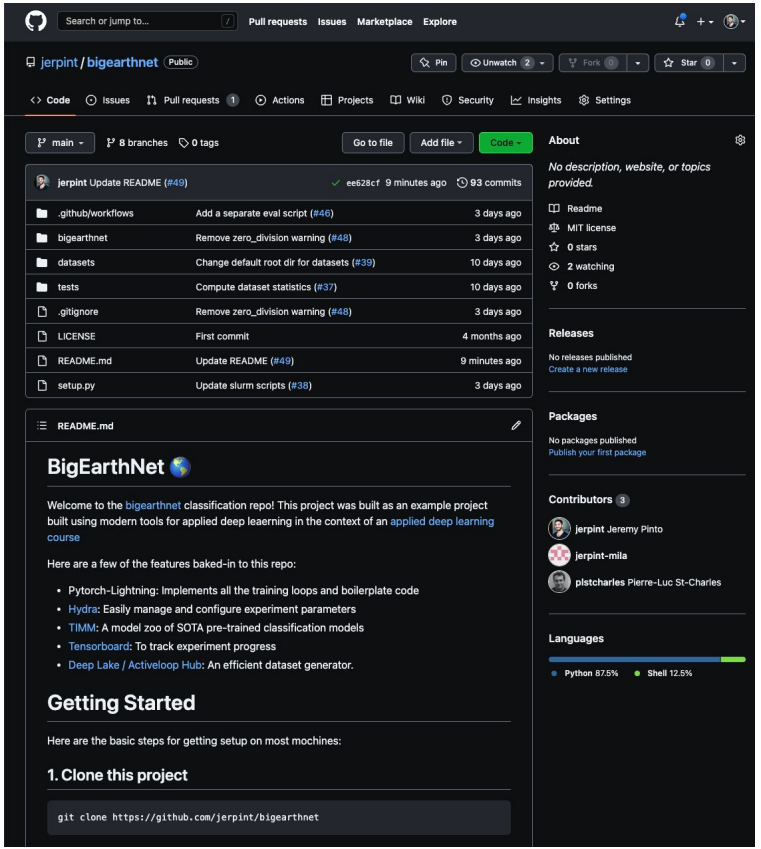
source

Code

Github

All of the code is made available on github:
https://github.com/jerpint/bigearthnet

To view the code within an **online IDE**
https://github.dev/jerpint/bigearthnet



IVADO Mila

Local Code Setup

pipenv

venv

To get the code setup, you need to **clone the code** and install the associated requirements.

We recommend using a **virtual environment** (conda, virtualenv, etc.) to isolate your project dependencies.

CONDA

```
# Download the source code
git clone https://github.com/jerpint/bigearthnet

# Create and activate your new conda environment
conda create -n bigearthnet python="3.8"
conda activate bigearthnet

# Install the project requirements
cd bigearthnet
pip install -e .
```


IVADO Mila

Test your setup

Once you have installed the requirements, you can test your setup with the following:

```
cd bigearthnet/bigearthnet/ # Go to the proper directory
python train.py # Run the code
```

It will take care of downloading the bigearthnet-mini dataset and train a baseline model over 3 epochs.

Setup.py

One handy feature is that we are installing our project as a **library**, we are not just installing requirements.

What this allows us to do is **import modules** anywhere in our codebase really easily.

setup.py

```python
from setuptools import setup, find_packages


setup(
    name="bigearthnet",
    version="0.0.1",
    packages=find_packages(include=["bigearthnet", "bigearthnet.*"]),
    python_requires=">=3.8",
    install_requires=[
        "gdown",
        "gitpython",
        "hub",
        "hydra-core>=1.2",
        "jupyter",
        "matplotlib",
        "numpy>=1.23",
        "pyyaml>=5.3",
        "pytest",
        "pytorch_lightning==1.6.4",
        "sklearn",
        "timm",
        "torch==1.11",
        "torch_tb_profiler",
        "tqdm",
    ],
    extras_require={
        "dev": ["opencv-python"],
    },
)

```

Colab

A sample colab to get setup and do some data analysis is provided along with the code.

Libraries

Pytorch Lightning

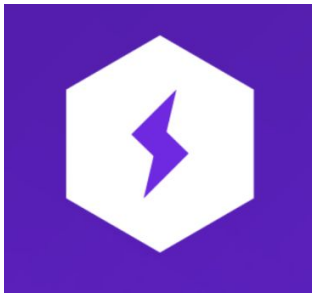
One of the main tools we use in the codebase is **pytorch lightning**.

It implements many useful routines for **training and deploying** pytorch models.

It **reduces** the need for boilerplate code and keeps the focus on the **research code** instead.

PyTorch

+



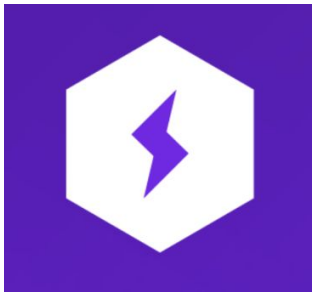
https://www.pytorchlightning.ai/

IVADO Mila

Pytorch Lightning

There are three main components in pytorch lightning:

- LightningModule
- DataModule
- Trainer

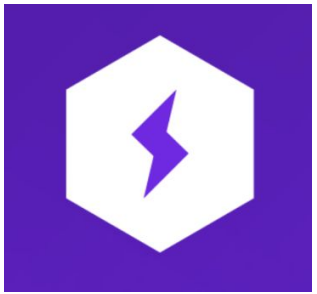


https://www.pytorchlightning.ai/

Pytorch Lightning

There are three main components in pytorch lightning:

- **LightningModule**
- DataModule
- Trainer



https://www.pytorchlightning.ai/

Pytorch Lightning

The model, training and validation steps, optimizers and loss get defined in **LightningModule** objects.

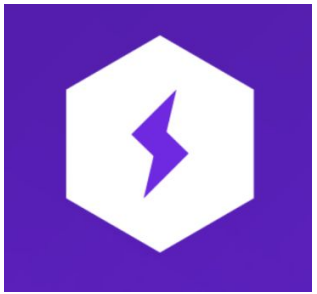
View the `LightningModule()` in our project:

```python
class BigEarthNetModule(pl.LightningModule):
    """Base class for Pytorch Lightning model."""

    def __init__(self, cfg: DictConfig):
        super().__init__()
        self.cfg = cfg
        self.model = instantiate(cfg.model)
        self.save_hyperparameters(cfg, logger=False)
        self.init_loss()

    def init_loss(self):...

    def on_train_start(self):...

    def configure_optimizers(self):...

    def _generic_step(self, batch, batch_idx):...

    def _generic_epoch_end(self, step_outputs):...

    def training_step(self, batch, batch_idx):...

    def training_epoch_end(self, training_step_outputs):...

    def validation_step(self, batch, batch_idx):...

    def validation_epoch_end(self, validation_step_outputs):...

    def test_step(self, batch, batch_idx):...

    def test_epoch_end(self, test_step_outputs):...

    def log_metrics(self, metrics: typing.Dict, split: str):...
```

Pytorch Lightning

There are three main components in pytorch lightning:

- LightningModule
- **DataModule**
- Trainer



https://www.pytorchlightning.ai/

Pytorch Lightning

The **DataModule()** allows us to organize and process our datasets and dataloaders.

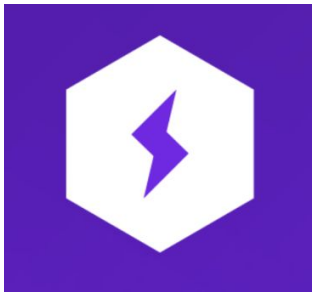
It also enables us to download datasets, process them, etc.

View the `DataModule()` in our project:

Pytorch Lightning

There are three main components in pytorch lightning:

- LightningModule
- DataModule
- **Trainer**



https://www.pytorchlightning.ai/

Pytorch Lightning

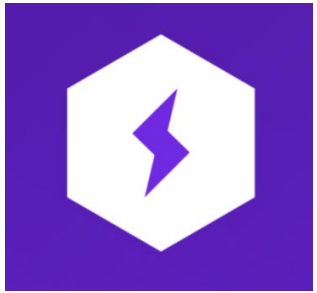
The **trainer()** takes in the **Datamodule** and **LightningModule** and orchestrates the training and validation logic.

```python
@hydra.main(config_path="configs", config_name="config", version_base="1.2")
def main(cfg: DictConfig):

    log.info("Beginning training...")

    # set seed if explicitly passed through CLI
    if cfg.experiment.get("seed"):
        log.info(f"Setting seed to: {cfg.experiment.seed}")
        set_seed(cfg.experiment.seed)

    # instantiate all objects from hydra configs
    model = BigEarthNetModule(cfg)
    datamodule = instantiate(cfg.datamodule)
    trainer = instantiate(cfg.trainer)

    # do the training
    datamodule.setup()
    trainer.fit(model, datamodule=datamodule)
    log.info("Training Done.")
```

Pytorch Lightning

The trainer also accepts **arbitrary callbacks**.
Many useful callbacks for training and deployment are available out of the box:

- Model checkpointing + early stopping
- Logger support (Wandb, tensorboard, etc.)
- Hardware support (GPU, TPU, fp16, DDP)
- Device profiling
- etc.

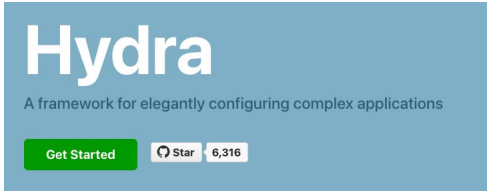


https://www.pytorchlightning.ai/

Hydra

Another tool used in the project is hydra.

It is a useful tool for managing arbitrarily complex yaml **configuration files**.



Hydra

All of the **default configurations** of experiments can be defined in hierarchical [configuration files](#).

! config.yaml ×

bigearthnet > configs > ! config.yaml

```yaml
datamodule:
  _target_: bigearthnet.datamodules.bigearthnet_datamodule.BigEarthNetDataModule
  dataset_dir: "${oc.env:HOME}/bigearthnet/datasets/" # root directory where to download the datasets
  dataset_name: "bigearthnet-mini" # One of bigearthnet-mini, bigearthnet-medium, bigearthnet-full
  batch_size: 16
  num_workers: 0
  transforms: ${transforms.obj}

optimizer:
  name: 'adam'  # adam or sgd
  lr: 0.0001  # learning rate

logger:
  # This project uses tensorboard as a logger.
  _target_: pytorch_lightning.loggers.TensorBoardLogger
  save_dir: "."

loss:
  class_weights: null # specify a path to the class_weights.json file if you want to re-balance the loss.

monitor:
  # Keeps track of this value to determine when to do model selection, patience, etc.
  name: "f1_score" # loss, f1_score, precision, recall
  mode: "max"  # min or max depending on metric, e.g. loss is min, f1_score is max
  patience: 10
```

Hydra

One nice feature using hydra is that CLI arguments are **easily defined and overridden** without needing any parsing code:

```
python train.py model="timm" ++model.model_name=resnet34
++model.pretrained="true" ++config.optimizer.name="adam"
++config.optimizer.lr="0.001" ++trainer.max_epochs="100"
```

ActiveLoop Deep Lake (Hub)

FFCV

🤗 Datasets

HDF5 TfRecord

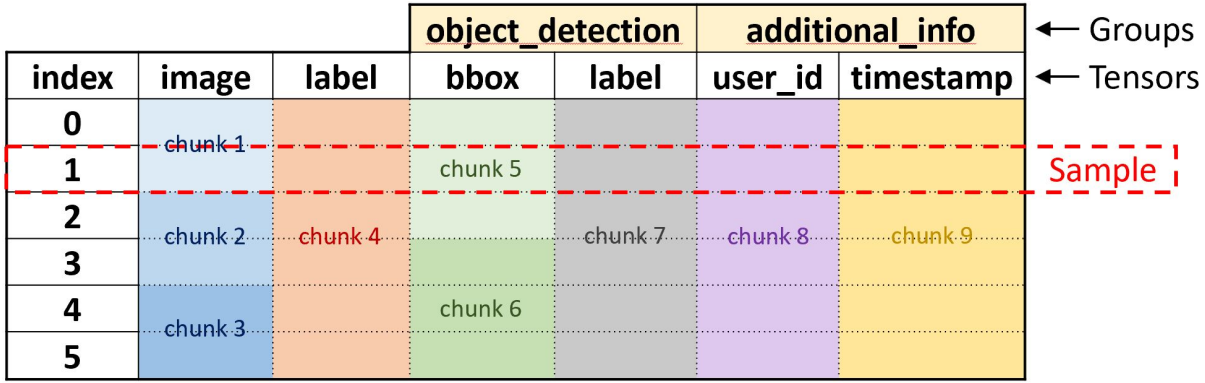
One **bottleneck** that often arises when training on large datasets is reading the data from disk and pre-processing it to be ready for our models.

To deal with this, we use Activeloop's [Deep Lake/Hub dataset format](#).

IVADO Mila

ActiveLoop Deep Lake (Hub)

Hub will chunk the data so that it is **easily accessible** during training, increasing I/O and limiting data loading bottlenecks.



ActiveLoop Deep Lake (Hub)

All **three datasets** (bigearthnet-mini, bigearthnet-medium and bigearthnet-full) were preprocessed using Hub. To specify which dataset to train on, simply override the `datamodule.dataset_name` attribute in your CLI:

```
python train.py ++datamodule.dataset_name=bigearthnet-medium
```

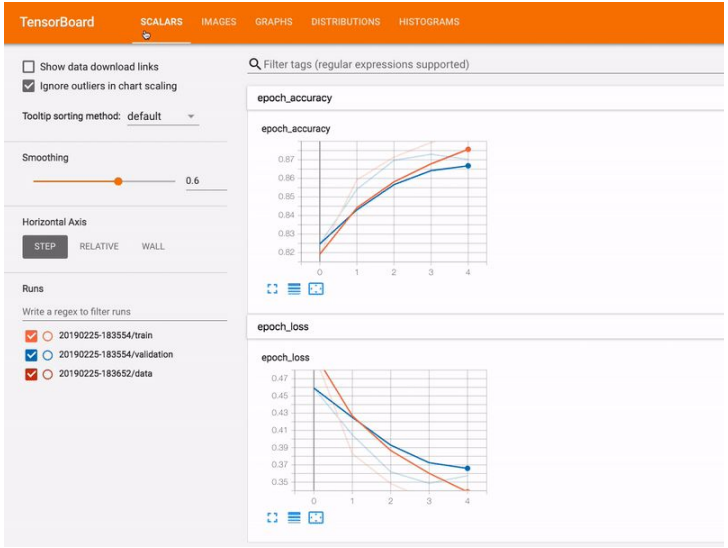
TensorBoard

W&B mlflow Aim comet

One crucial aspect of training many models on a dataset is **experiment tracking**.

The experiment tracking library we use in this project is **TensorBoard**.

It **integrates** nicely with pytorch-lightning and has a bunch of handy built-in features.



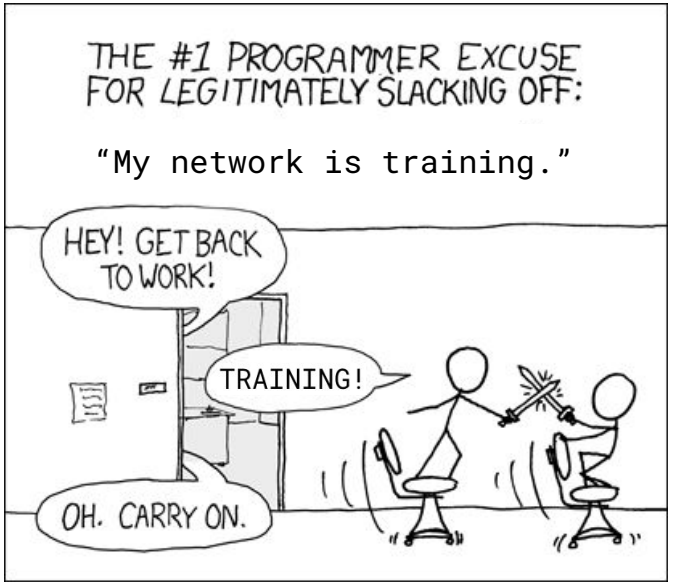
Source

IVADO Mila

TensorBoard

More importantly, experiment trackers allows us to **step away** from the computer while the models train.

They will **centralize our results** and make them easier to view at a later time.



TensorBoard

To view tensorboard on your own local experiments, train a model and point tensorboard to the `outputs` directory:

```
python train.py # Train a model
tensorboard --logdir outputs/ # Launch a local instance of tensorboard and navigate to
http://localhost:6006/ in your browser
```

timm

Pytorch Image Models (timm) is a model zoo of **state-of-the-art** pre-trained computer-vision models.

The list of available models is exhaustive.

```
1 !pip install -q timm
2
3 import timm
4 display(timm.list_models("*vit*"))
```

```
--NORMAL--
'vit_base_patch8_224_dino',
'vit_base_patch8_224_in21k',
'vit_base_patch16_18x2_224',
'vit_base_patch16_224',
'vit_base_patch16_224_dino',
'vit_base_patch16_224_in21k',
'vit_base_patch16_224_miil',
'vit_base_patch16_224_miil_in21k',
'vit_base_patch16_224_sam',
'vit_base_patch16_384',
'vit_base_patch16_plus_240',
'vit_base_patch16_rpn_224',
'vit_base_patch32_224',
'vit_base_patch32_224_in21k',
```

colab

timm

Using any of their existing models in our code is as easy as:

```
python train.py model="timm" ++model.model_name=resnet101 ++model.pretrained="true"
```

Baseline

Baseline Model

We will start by implementing the same **baseline model** as in the BigEarthNet [dataset publication]:

> *"we selected a shallow CNN architecture, which consists of **three convolutional layers** with 32, 32 and 64 filters having 5 × 5, 5 × 5 and 3 × 3 filter sizes, respectively.*
>
> *We added **one fully connected (FC) layer and one classification layer** to the output of last convolutional layer. In all convolution operations, **zero padding** was used. We also applied **max-pooling** between layers."*

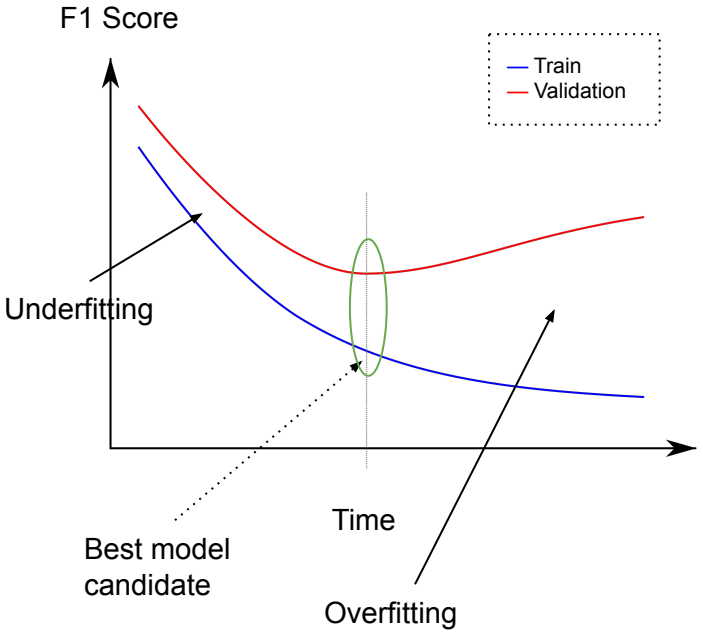
```python
class Baseline(torch.nn.Module):  # pragma: no cover
    """Baseline Model Class.

    Inherits from the given framework's model class. This is a simple MLP model.
    """

    def __init__(
        self,
        num_classes: int,
        hidden_dim: int,
        model_name: str,
        pretrained: bool,
    ):
        """__init__.

        Args:
            hyper_params (dict): hyper parameters from the config file.
        """
        super(Baseline, self).__init__()
        self.model_name = model_name
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 32, 5, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 32, 5, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.flatten = nn.Flatten()
        self.mlp_layers = nn.Sequential(
            nn.Linear(
                14400,
                hidden_dim,
            ),  # The input size for the linear layer is determined by the previous operations
            nn.ReLU(),
            nn.Linear(
                hidden_dim, num_classes
            ),  # Here we get exactly num_classes logits at the output
        )
        assert pretrained is False, "No pretrained models exist for the baseline"

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.flatten(x)  # Flatten is necessary to pass from CNNs to MLP
        x = self.mlp_layers(x)
        return x
```

model definition

Model Selection

To select our best model during training, we checkpoint the model which scored the best F1 score on the **validation set**.

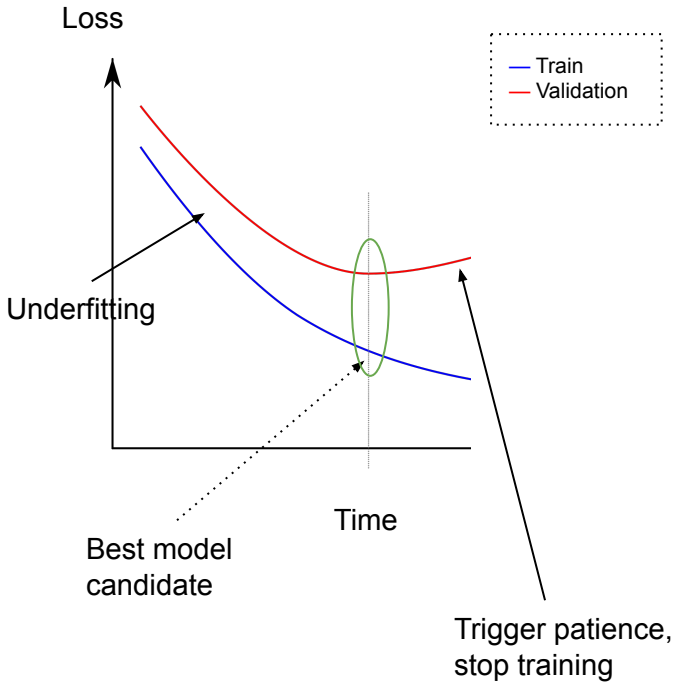
This is done by **monitoring** the f1 score throughout training, and easily implemented in pytorch lightning via **callbacks**.



Model Selection

We can also set a **patience** to our model.

If after N epochs, we do not see an improvement, we can simply **halt** the training.



Hyper-Parameters

Deep learning is **inherently empirical**. How can we determine:

- How many layers/**dimensions** to use?
- Which **optimizer** to use?
- What is the optimal **learning rate**?

All of these non-trainable parameters are called **hyper-parameters**.

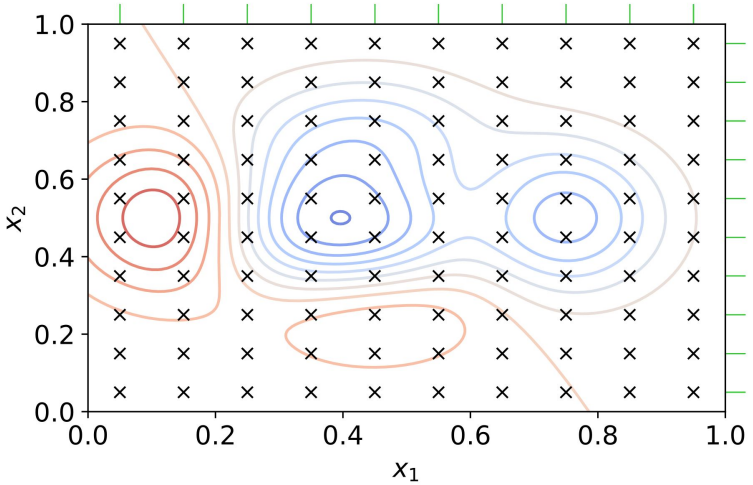
```python
class Baseline(torch.nn.Module):  # pragma: no cover
    """Baseline Model Class.

    Inherits from the given framework's model class. This is a simple MLP model.
    """

    def __init__(
        self,
        num_classes: int,
        hidden_dim: int,
        model_name: str,
        pretrained: bool,
    ):
        """__init__.

        Args:
            hyper_params (dict): hyper parameters from the config file.
        """
        super(Baseline, self).__init__()
        self.model_name = model_name
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 32, 5, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 32, 5, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.flatten = nn.Flatten()
        self.mlp_layers = nn.Sequential(
            nn.Linear(
                14400,
                hidden_dim,
            ),  # The input size for the linear layer is determined by the previous operations
            nn.ReLU(),
            nn.Linear(
                hidden_dim, num_classes
            ),  # Here we get exactly num_classes logits at the output
        )
        assert pretrained is False, "No pretrained models exist for the baseline"

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.flatten(x)  # Flatten is necessary to pass from CNNs to MLP
        x = self.mlp_layers(x)
        return x
```

[model definition]

Hyper-Parameter Search

One approach is to perform a **grid-search** over the hyper parameter space.

We select the best performing model on the **validation set** to determine the “best” hyper-parameters.



Hyper-Parameter Search

One very handy feature in hydra is the ability to launch **multiple experiments** with the `--multirun` flag:

```
python train.py --multirun
model="baseline"
++datamodule.dataset_name="bigearthnet-medium"
++model.hidden_dim="256","512"
++config.optimizer.name="adam","sgd"
++config.optimizer.lr="0.01","0.001","0.0001","0.00001"
++trainer.max_epochs="100"
```

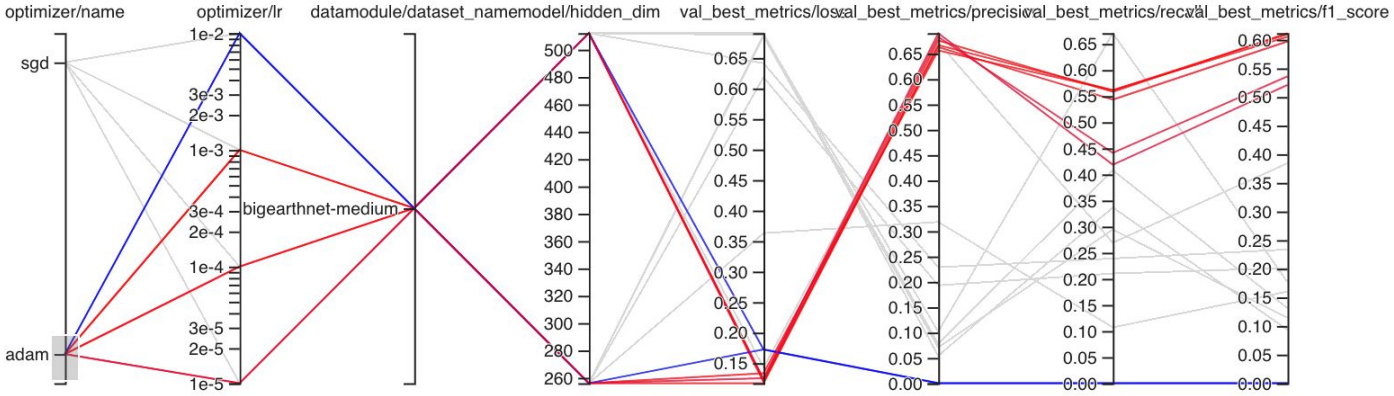
Pre-Trained Models

Similarly, we can train on **pre-trained models** and compare their performance to our baseline.

```
python train.py --multirun
++datamodule.dataset_name="bigearthnet-medium"
model="timm" ++model.model_name="resnet50","resnet101","vit_base_patch16_224"
++model.pretrained="true","false"
++trainer.max_epochs="100"
```

Hyper-Parameter Search

After running our **hyper-parameter** search, we can look for trends in our various hyper-parameter choices.



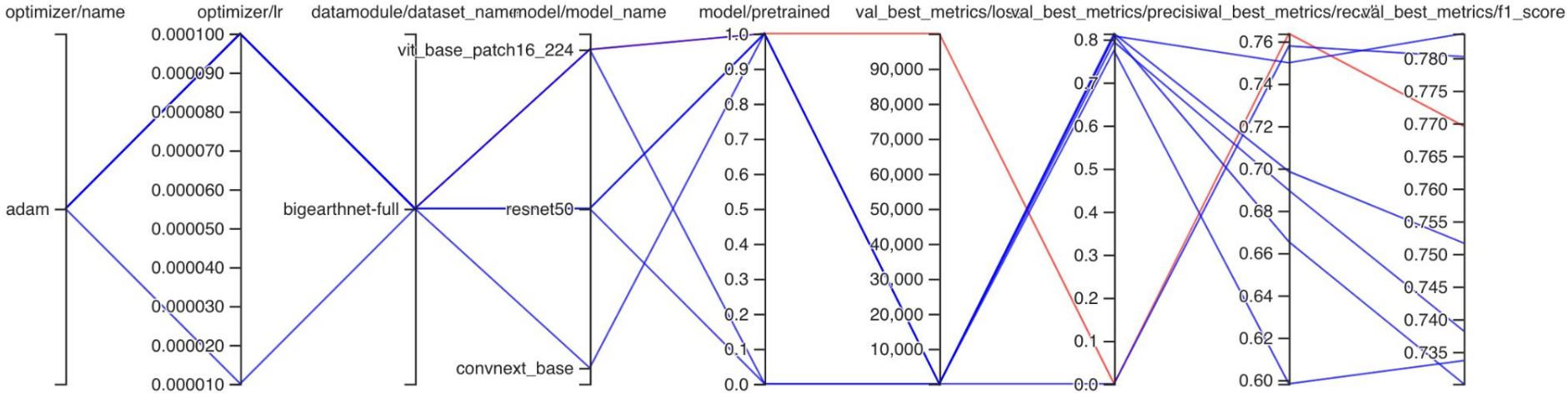
Tensorboard

Tensorboard demo.

Model Evaluation

Evaluation

After a hyper-parameter search, select the model(s) that performed best on the **validation metric** and perform an evaluation on the test set. In general, it is a **good practice** to use the test set **sparingly**.



Tensorboard Exp. Link

You can view the experiment on tensorboard

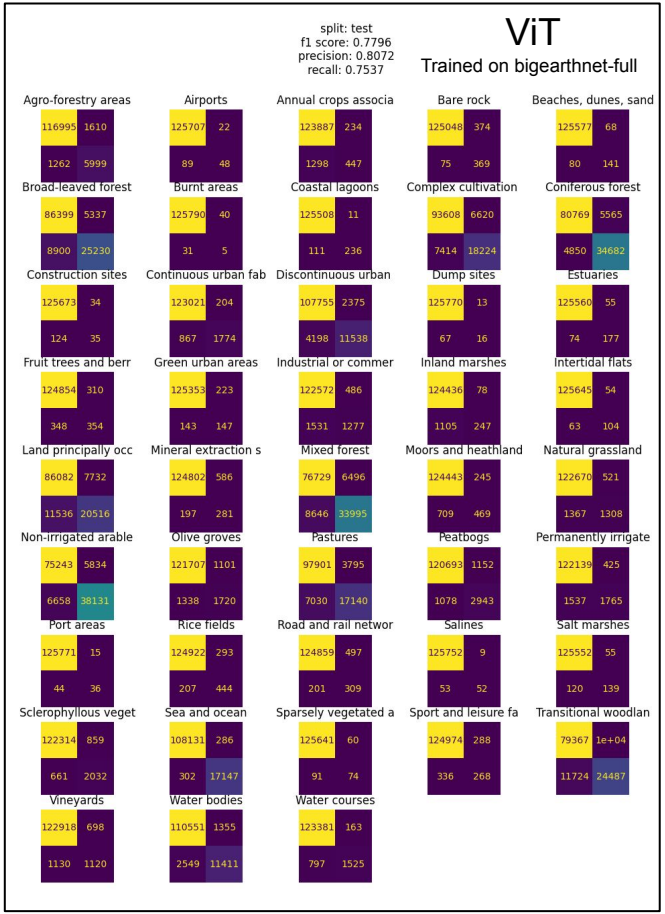
Test set results

Here, we re-trained the best models from the bigearthnet-medium to the bigearthnet-full dataset. Unsurprisingly, **more data** helped improve the overall **F1-score**.

Model	Training Data	Bigearthnet-full test set		
		Precision	Recall	F1 Score
ConvNext	bigearthnet-medium	0.765	0.687	0.724
ViT	bigearthnet-medium	0.784	0.713	0.747
ConvNext	bigearthnet-full	0.812	0.755	**0.782**
ViT	bigearthnet-full	0.807	0.754	0.780

Results

Similar performance for both ConvNext and ViT.



Results

Notice an increased performance on the **unbalanced classes** (green urban areas, inland marshes)

Best Practices

Reproducibility

Reproducing key experiments can be **crucial** in certain circumstances.

Keeping track of which **version** of the code and libraries that were used is crucial.

Another good practice is to set **seeds** if you want **exact** results to be reproducible.

```python
class ReproducibilityLogging(Callback):
    """Log experiment details for reproducibility.

    This will pring the git hash, branch name, dependencies and
    omegaconf config to the log and save the omegaconf config to disk.
    """

    @staticmethod
    def parse_exp_details(cfg: DictConfig):  # pragma: no cover
        """Will parse the experiment details to a readable format for logging.

        :param cfg: (OmegaConf) The main config for the experiment.
        """
        # Log and save the config used for reproducibility
        script_path = get_original_cwd()
        git_hash, git_branch_name = get_git_info(script_path)
        hostname = socket.gethostname()
        dependencies = freeze.freeze()
        dependencies_str = "\n".join([d for d in dependencies])
        details = f"""
                    config: {OmegaConf.to_yaml(cfg)}
                    hostname: {hostname}
                    git code hash: {git_hash}
                    git branch name: {git_branch_name}
                    dependencies: {dependencies_str}
                    """
        return details

    def log_exp_info(self, trainer, pl_module):
        """Log info like the git branch, hash, dependencies, etc."""
        cfg = pl_module.cfg
        exp_details = self.parse_exp_details(cfg)
        log.info("Experiment info:" + exp_details + "\n")

        # dump the omegaconf config for reproducibility
        output_dir = os.path.join(trainer.logger.log_dir) if trainer.logger else "."
        if not os.path.isdir(output_dir):
            os.makedirs(output_dir)

        OmegaConf.save(cfg, os.path.join(output_dir, "exp_config.yaml"))

    def on_train_start(self, trainer, pl_module):
        self.log_exp_info(trainer, pl_module)

    def on_test_start(self, trainer, pl_module):
        self.log_exp_info(trainer, pl_module)
```

Debugging Dataset

Using a **small dataset** can make local debugging easier and more efficient. Mock data is a good alternative if data is sensitive.

Quickly implement new features without always needing to provision **expensive** GPUs and wait for long training times.

It can also be used in **CI/CD** pipelines to test that everything is running as intended.

```yaml
name: ci-pipeline
on:
  # Trigger the workflow on push or pull request,
  # but only for the main branch
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
  ci-pipeline:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v3
      - name: black linter
        uses: psf/black@stable
      - name: python-3.8
        uses: actions/setup-python@v2
        with:
          python-version: 3.8
      - name: install-dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -e .
      - name: unit_tests
        run: pytest tests/*
      - name: end_to_end_run
        run: |
          cd bigearthnet
          python train.py ++datamodule.dataset_dir=$GITHUB_WORKSP
```

Intermediate Dataset

When dealing with large datasets, having an **intermediate dataset** can help testing out new model architectures, intuitions, hyper-parameters etc.

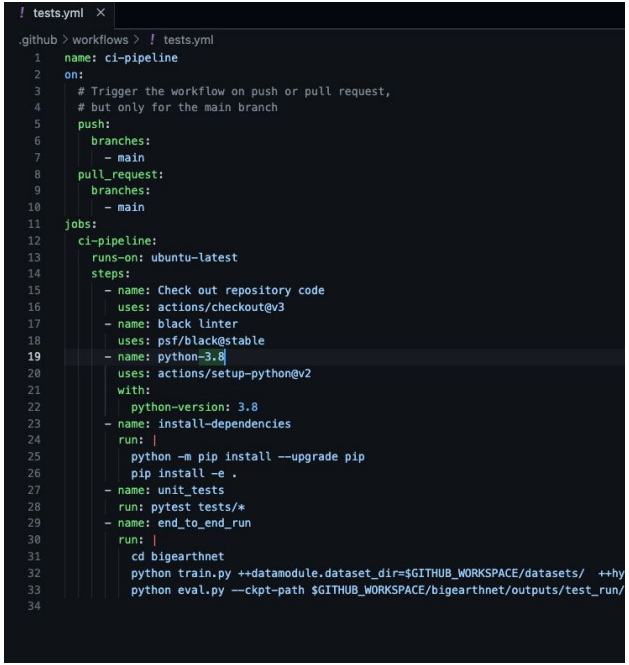
If it doesn’t work on the medium dataset, it **probably won’t work** on the large dataset.

Larger experiments can then be run **pragmatically** once the pipeline is well understood.

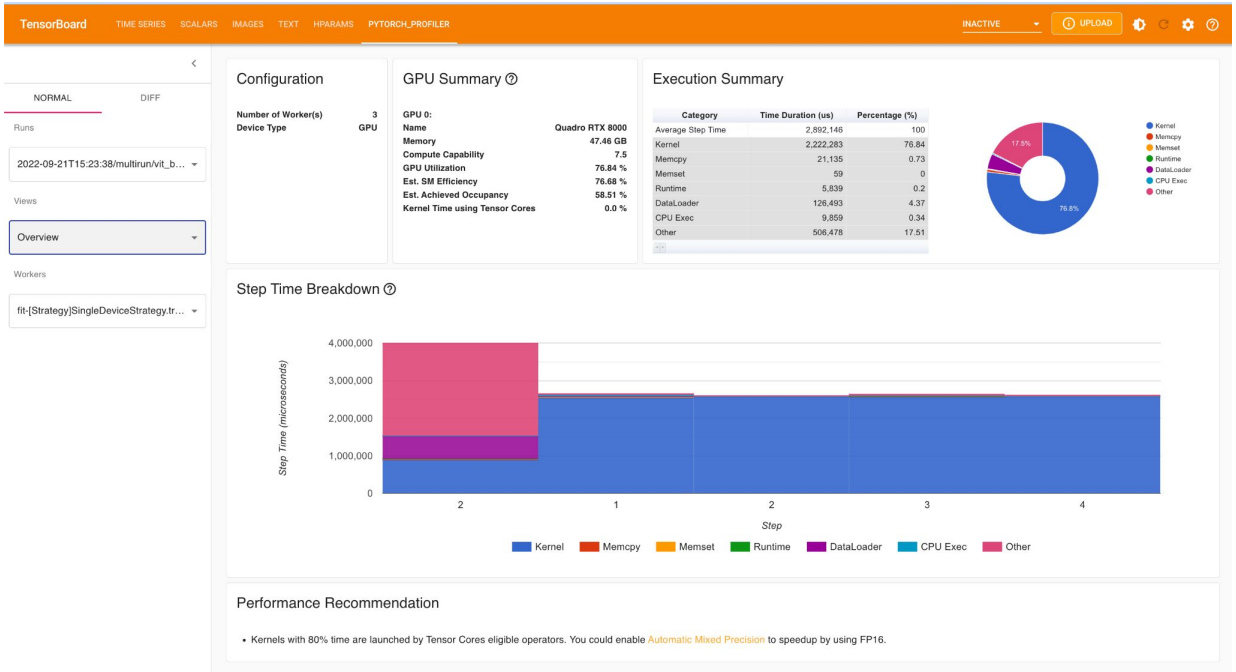
CI/CD and Unit Tests

Platforms like github/gitlab/etc. allow running **unit tests** and other scripts before merging a new branch.

Training a model **end to end** on a small dataset can be a great way to check if new code doesn't break anything in our pipelines.



Profiling GPU Usage

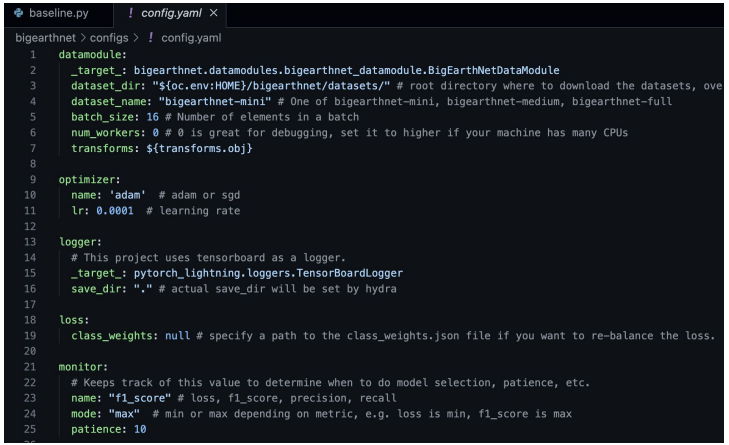


Class Imbalance

Class Imbalance

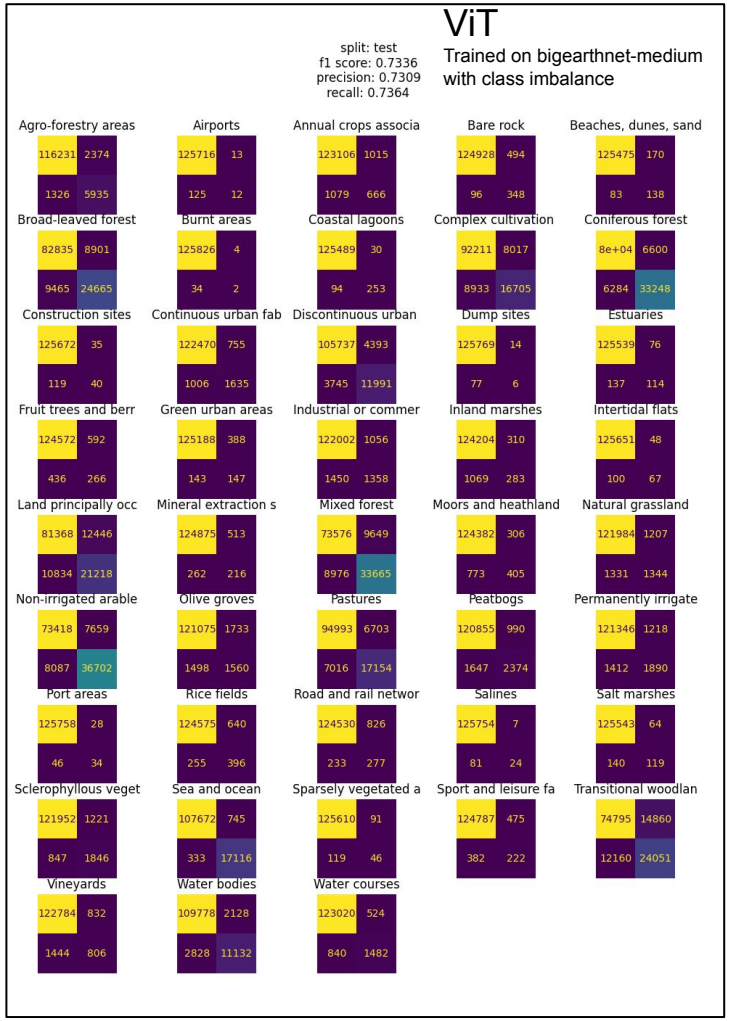
In pytorch, the simplest way to deal with **class imbalance** is setting the **class weights** in the loss function.

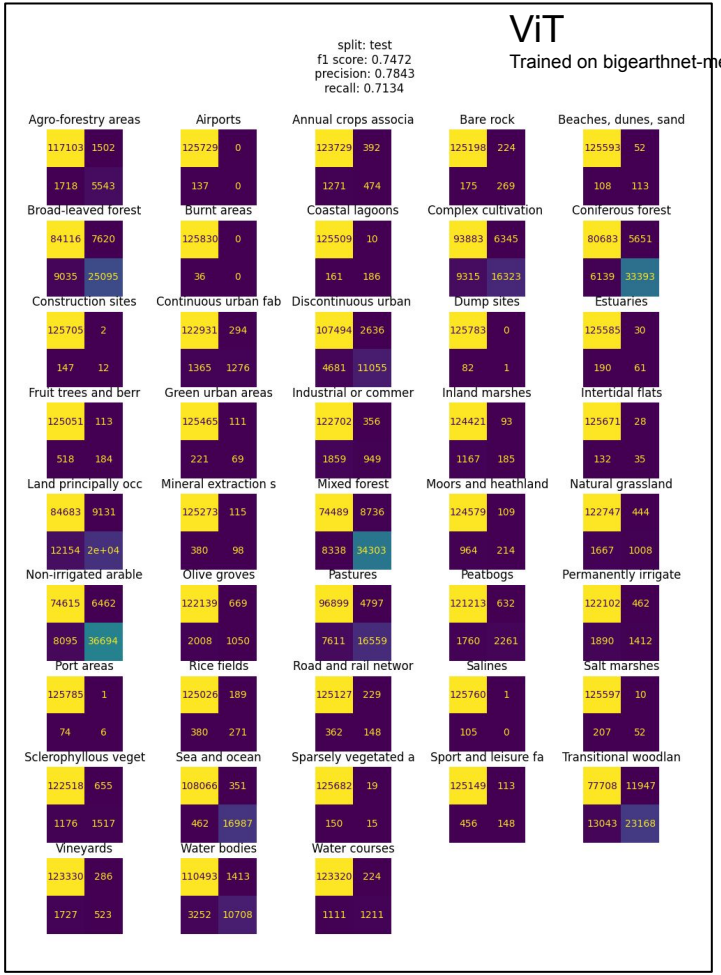
From the pytorch docs: “It’s possible to **trade off** recall and precision by adding weights to positive examples.”



Decreased precision: More false positives

Increased recall: More true events get detected



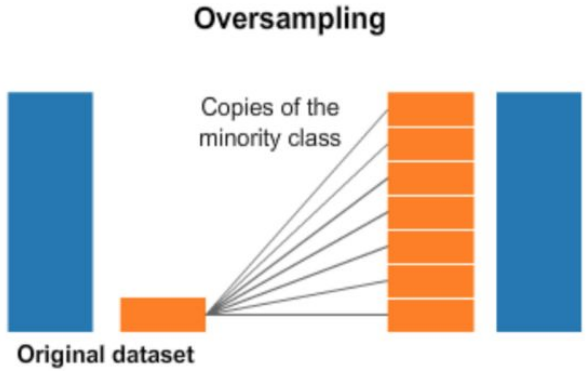


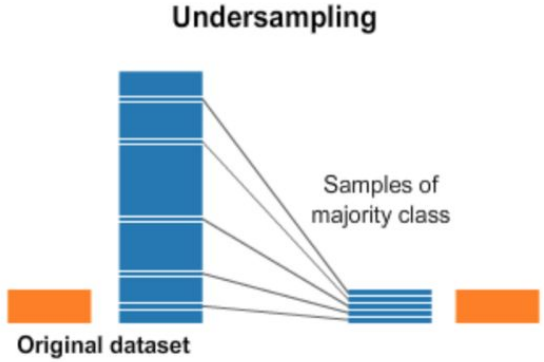
Class Imbalance

Another way is to oversample/undersample data.

Oversampling can sometimes lead to **overfitting**.

Undersampling can lead to **underfitting**.





source

Focal Loss

Custom loss functions, like **focal loss**, can help put more emphasis on “hard samples” in a principled way:

“The modulating factor reduces the loss contribution from easy examples and extends the range in which an example receives low loss.”

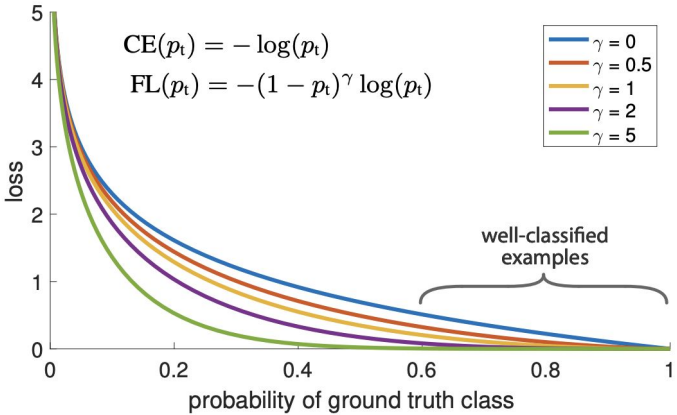


Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^{\gamma}$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.

source

pytorch implementation

Questions?

jeremy.pinto@mila.quebec
pierreluc.stcharles@mila.quebec