

EagleEyes_SouthAfrica

January 6, 2022

1 AI4Food Security South Africa

Team EagleEyes:

Christina Bukas, HMGU

Frauke Albrecht, DKRZ

Caroline Arnold, DKRZ

We would also like to thank Elisabeth Georgii from HMGU for her contribution in literature research and useful insights when discussing methods

1.1 Introduction

The aim of this challenge is to identify crop types in-season using Planet, Sentinel-1, and Sentinel-2 data. Field boundaries and crop type labels are given as ground truth data.

1.2 Setup

In the following cell, all modules are imported that are necessary in this notebook.

```
[1]: import os
import copy

import h5py
import numpy as np
import geopandas as gpd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset
```

1.3 Data exploration

1.3.1 Preprocessing

Data from Planet, Sentinel-1, and Sentinel-2 satellites are given for a region in South Africa. We process the datasets individually, building on the routines that were provided through the starter notebook. The workflow is described here for Planet data.

1. Rasterize the satellite images using the provided routines, for each field save a zip file containing the time series of bands and the mask
2. Load the field ID and apply custom data transform. Available transforms are: spatial average, crop image, extract fixed number of pixels
3. Save as hdf5 files for use in training

The full preprocessing module is included in `dataloaders/preprocessor.py`, with the dataset readers in `dataloaders/custom_{planet,sentinel_1,sentinel_2}_reader.py` and the image transforms in `dataloaders/custom_data_transform.py`.

We generated the training data by running

```
python preprocessor.py --region south-africa --data-source planet
--t-random-extraction 640 --n-processes 64 --target-sub-dir extracted-640
--overwrite
```

```
python preprocessor.py --region south-africa --data-source sentinel-1
--t-random-extraction 640 --n-processes 64 --target-sub-dir extracted-640
--overwrite
```

```
python preprocessor.py --region south-africa --data-source sentinel-2
--t-random-extraction 640 --n-processes 64 --target-sub-dir extracted-640
--overwrite
```

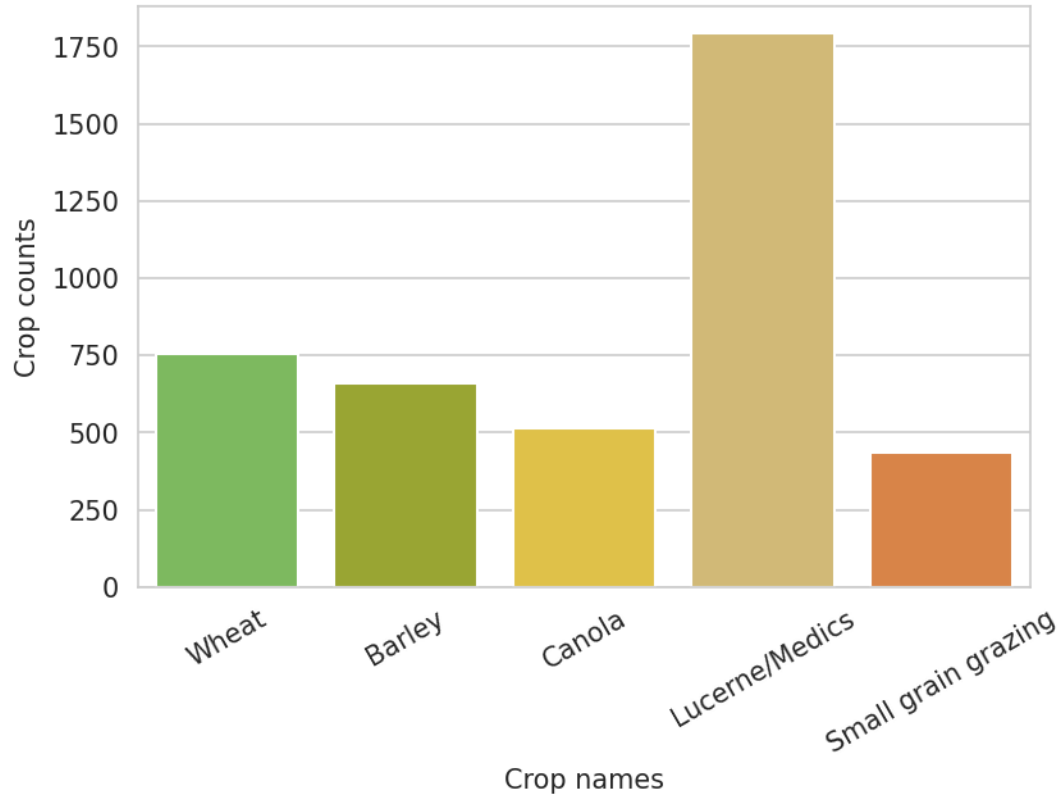
This extracts 640 pixels at random from a field, taking into account only the valid pixels as given by the mask. We resample if there are less than 640 pixels are available. Then, out of each field, we create 10 training samples, containing 64 extracted pixels each. This routine is execute in the notebook `DataInflation.ipynb`. Thus, we arrive at a total count of 41430 samples that can be used in training / validation. For the test set, 64 pixels are extracted at random per field.

1.3.2 Training and validation

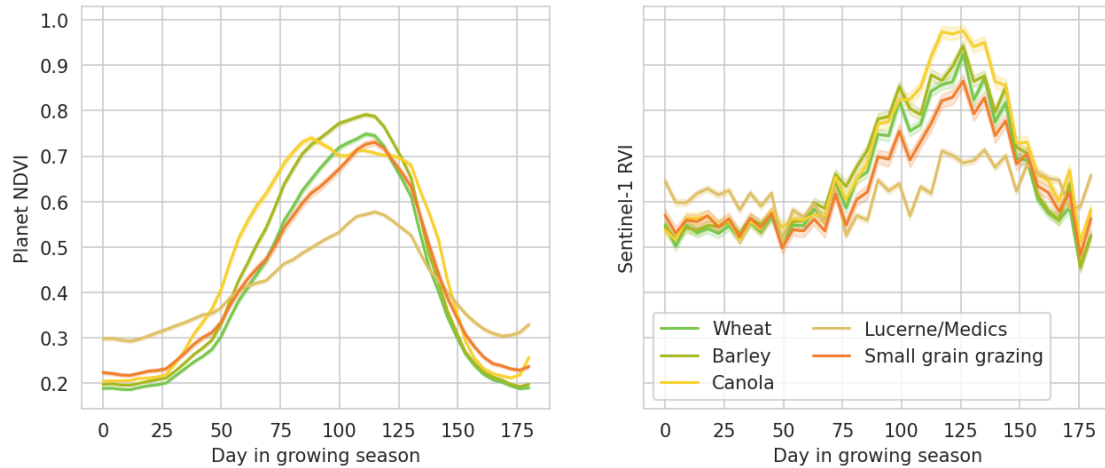
We use 10-fold cross validation, dividing the labeled data into training and validation set by field ID.

1.3.3 Exploration

The crop types are not evenly distributed:



We calculate the normalized vegetation index NDVI as the normalized difference of infrared and red planet data bands. Below (left) the average NDVI is shown throughout the growing season for the different crop types. For Sentinel-1 data, we calculate the Radar Vegetation Index (right).



1.4 PSELTAE Model

The model used is described in detail in “Satellite Image Time Series Classification With Pixel-Set Encoders and Temporal Self-Attention”, Garnot. et al. (2020) and the code has been adapted from <https://github.com/VSainteuf/pytorch-psetae>. The model consists of a Spatial and a Temporal encoder. As spatial encoder a Pixel-Set Encoder is used, which uses as input a random set of pixels

from each crop field. As Temporal Encoder, an attention based Neural Network is used. We train a combined model with Planet and Sentinel-1 data and therefore use two separate PseLTae models for each dataset whose outputs are combined in the final decoder layer. Combination with an additional PseLTae model for Sentinel-2 data was explored, but did not improve the classification accuracy.

In order to train the model the datasets have been created by the following code:

```
[2]: #!/usr/bin/env python

from torch.utils.data import Dataset
import h5py
import os
import numpy as np
import geopandas as gpd
from torch import randn
import time
from scipy.interpolate import splrep, splev
from scipy.signal import savgol_filter

class EarthObservationDataset(Dataset):
    """
    Parent class for Earth Observation Datasets

    Preprocessed data is loaded from path:

    -- in args namespace --
    dev_data_dir      : file path on mistral, {germany, south africa}
    input_data[0]     : data source {planet, sentinel-1, sentinel-2}
    input_data_type    : {extracted, extracted-640}
    split             : {train, test}

    If args.include_extras, the crop_area and crop_len are included for each_
    →sample
    """

    def __init__(self, args):
        super().__init__()
        self.args = args

        self.h5_file = h5py.File(os.path.join(args.dev_data_dir, args.
    →input_data[0], args.input_data_type, f'{args.split}_data.h5'), 'r')

        self.X = self.h5_file['image_stack'][:].astype(np.float32)
        self.mask = self.h5_file['mask'][:].astype(bool)
        self.fid = self.h5_file['fid'][:]
        self.labels = self.h5_file['label'][:]
```

```

        self.labels = self.labels - 1 # generated datafiles with classes from 1 .
→... k --> 0 ... k-1
        if np.sum(np.isnan(self.X)) > 0:
            print('WARNING: Filled NaNs and INFs with 0 in ', os.path.join(args.
→dev_data_dir, args.input_data[0], args.input_data_type, f'{args.split}_data.
→h5'))
            self.X = np.nan_to_num(self.X, nan=0, posinf=0, neginf=0)

        if args.include_extras:
            labels_path = os.path.join(args.dev_data_dir, 'labels_combined.
→geojson')
            print('Adding extra features from ', labels_path)
            extras = gpd.read_file(labels_path)

            crop_area = []
            crop_len = []

            extras_fid = extras["fid"].values
            extras_crop_area = extras["NORMALIZED_SHAPE_AREA"].values
            extras_crop_len = extras["NORMALIZED_SHAPE_LEN"].values

            for ii, ffid in enumerate(self.fid):
                ix = np.where(extras_fid==ffid)[0][0]
                crop_area.append(extras_crop_area[ix])
                crop_len.append(extras_crop_len[ix])

                if ii%(len(self.fid)//20) == 0:
                    print(f'... finished {ii:8d}/{len(self.fid):8d} entries ({ii/
→len(self.fid)*100:.1f} %)')

            self.extra_features = np.array([crop_area, crop_len]).T
        else:
            self.extra_features = None

        # remove a fully masked sample from the Germany training data
        if self.args.nr_classes == 9 and self.args.split == 'train':
            if self.args.input_data_type == 'extracted':
                bad_idx = [1225]
            elif self.args.input_data_type == 'extracted-640':
                bad_idx = [12250, 12251, 12252, 12253, 12254, 12255, 12256,
→12257, 12258, 12259]
            else:
                bad_idx = []
            self.X = np.delete(self.X, bad_idx, axis=0)
            self.mask = np.delete(self.mask, bad_idx, axis=0)
            self.fid = np.delete(self.fid, bad_idx, axis=0)
            self.labels = np.delete(self.labels, bad_idx, axis=0)

```

```

def __len__(self):
    return len(self.labels)

def __getitem__(self, idx):
    X = self.X[idx]
    label = self.labels[idx]
    mask = self.mask[idx]
    fid = self.fid[idx]
    if self.extra_features is not None:
        extra_f = self.extra_features[idx]
    else: extra_f = np.zeros_like(1)

    return (X, mask, fid, extra_f), label

class Sentinel1Dataset(EarthObservationDataset):
    '''
    Sentinel 1 Dataset

    If args.nri, calculates indices:

    Normalized radar vegetation index (RVI)

    If args.drop_channels_sentinel1, drop all bands

    If args.savgol_filter, smooth the RVI with a Savitzky Golay filter

    If args.split_nri, create two separate RVI channels for each observation_
→angle
    '''
    def __init__(self, args):
        super().__init__(args)

        self.X = self.X[:, :, :2, :] # only VV and VH (2) is the angle

        # ! -- Germany test data has different length, cut to to train data_
→length
        if args.nr_classes == 9 and args.split == 'test':
            print('Sentinel-1 shape before cut: ', self.X.shape)
            self.X = self.X[:, 1:-1]
            print('Sentinel-1 shape after cut (test set germany): ', self.X.
→shape)

        if args.nri:
            nri = Sentinel1Dataset._calc_rvi(self.X, self.args.savgol_filter)
            nri = np.expand_dims(nri, axis=2) # changed axis from 1 to 2
            if args.drop_channels or args.drop_channels_sentinel1:

```

```

        if args.split_nri: # split into two channels for different angles
            if self.args.savgol_filter:
                raise ValueError("Do not use Savitzky Golay filter and_
→NRI split together")
            nri_odd = nri[:, ::2]
            nri_even = nri[:, 1::2]
            nri_min_len = min(nri_odd.shape[1], nri_even.shape[1])
            self.X = np.concatenate([nri_odd[:, :nri_min_len], nri_even[
→, :nri_min_len]], axis=2)
        else:
            self.X = nri
    else:
        self.X = np.concatenate([self.X, nri], axis=2)

    print('Final shape for Sentinel-1 image stack: ', self.X.shape)

    '''
    # normalization of datasets min-max
    xmin=np.min(self.X, axis=(0,1,3))
    xmax=np.max(self.X, axis=(0,1,3))
    for i in range(self.X.shape[2]):
        self.X[:, :, i, :] = (self.X[:, :, i, :] - xmin[i])/(xmax[i] - xmin[i])
    '''

    @staticmethod
    def _calc_rvi(X, rvi_filter=False):
        VV = X[:, :, 0, :]
        VH = X[:, :, 1, :]
        dop = (VV/(VV+VH))
        m = 1 - dop
        radar_vegetation_index = (np.sqrt(dop))*((4*(VH))/(VV+VH))

        eps = 1e-9 # avoid zero values
        radar_vegetation_index = np.nan_to_num(radar_vegetation_index, nan=eps,
→posinf=eps, neginf=eps)

        if not rvi_filter:
            return radar_vegetation_index

        start_time = time.time()

        print('Start to apply Savitzky Golay Filter to Sentinel-1 RVI')
        for i in range(radar_vegetation_index.shape[0]):
            for j in range(radar_vegetation_index.shape[-1]):
                tmp = radar_vegetation_index[i, :, j]

                smooth_rvi = savgol_filter(tmp, 15, 3)

```

```

        radar_vegetation_index[i, :, j] = smooth_rvi

    print(f'Applied Savitzky Golay Filter to Sentinel-1 RVI in {time.time() -
→ start_time:.1f} seconds')

    return radar_vegetation_index

class PlanetDataset(EarthObservationDataset):
    '''
    Planet Dataset

    If args.ndvi, calculates indices:

    Normalized difference vegetation index (NDVI)

    If args.drop_channels, drop all bands

    '''

    def __init__(self, args):
        super().__init__(args)
        if args.ndvi:
            ndvi = PlanetDataset._calc_ndvi(self.X)
            ndvi = np.expand_dims(ndvi, axis=2) # changed axis from 1 to 2
            if args.drop_channels:
                self.X = ndvi
            else:
                self.X = np.concatenate([self.X, ndvi], axis=2) # changed axis
→from 1 to 2

        if args.nr_classes == 9 and args.vegetation_period: # Germany
            ix_train_start = 83 # defined as the minimum of NDVI
            ix_test_start  = 90 # such that NDVI maxima match
            vg_length      = 180 # length of the vegetation period

            if args.split == 'train':
                self.X = self.X[:, ix_train_start:ix_train_start + vg_length]
            elif args.split == 'test':
                self.X = self.X[:, ix_test_start:ix_test_start + vg_length]

        print('Final shape for Planet image stack', self.X.shape)

    '''
    # normalization of datasets min-max
    xmin=np.min(self.X, axis=(0,1,3))

```



```

        xmax=np.max(self.X, axis=(0,1,3))
        for i in range(self.X.shape[2]):
            self.X[:, :, i, :] = (self.X[:, :, i, :] - xmin[i])/(xmax[i] - xmin[i])
        '''

    @staticmethod
    def _calc_ndvi(X):
        '''
        Calculate the normalized vegetation index

        NDVI = (NIR - RED) / (NIR + RED)

        '''
        #print(X.shape) #(4143, 244, 4, 64)
        nir = X[:, :, 3, :] # X[:, 3]
        red = X[:, :, 2, :] # X[:, 2]
        ndvi = (nir - red) / (nir + red)
        ndvi = np.nan_to_num(ndvi)
        return ndvi

class CombinedDataset(Dataset):
    '''
    Class for a combined dataset, holds PlanetDataset, Sentinel1Dataset,
    →Sentinel2Dataset
    as specified by args.input_data
    '''
    def __init__(self, args):
        super().__init__()
        self.datasets = []
        self.input_data = args.input_data.copy()
        for input_data in self.input_data:
            if input_data=='planet':
                args.input_data = ['planet']
                planet_dataset = PlanetDataset(args)
                self.datasets.append(planet_dataset)
            elif input_data=='planet-5':
                args.input_data = ['planet-5']
                planet5_dataset = PlanetDataset(args)
                self.datasets.append(planet5_dataset)

            elif input_data=='sentinel-1':
                args.input_data = ['sentinel-1']
                sentinel1_dataset = Sentinel1Dataset(args)
                self.datasets.append(sentinel1_dataset)
            elif input_data=='sentinel-2':

```

```

        args.input_data = ['sentinel-2']
        sentinel2_dataset = Sentinel2Dataset(args)
        self.datasets.append(sentinel2_dataset)
    args.input_data = self.input_data
    for i in range(1, len(self.datasets)):
        print('Assert dataset shape match', i-1, i)
        assert (self.datasets[i-1].fid==self.datasets[i].fid).all(), 's1, s2_
→and/or planet not sorted correctly'

    def __len__(self):
        return len(self.datasets[0].labels)

    def __getitem__(self, idx):
        return tuple(d[idx] for d in self.datasets)

class AddGaussianNoise(object):
    """
    Add Gaussian noise to a sample
    """
    def __init__(self, mean=0., std=1.):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean,
→self.std)

```

In the following the model components are listed:

Pixel-Set-Encoder

```

[3]: class PixelSetEncoder(nn.Module):
        def __init__(self, input_dim, mlp1=[10, 32, 64], pooling='mean_std',
→mlp2=[64, 128],
            with_extra=True, extra_size=4):
            """
            Pixel-set encoder.
            Args:
                input_dim (int): Number of channels of the input tensors
                mlp1 (list): Dimensions of the successive feature spaces of MLP1
                pooling (str): Pixel-embedding pooling strategy, can be chosen in_
→('mean', 'std', 'max', 'min')
                or any underscore-separated combination thereof.
                mlp2 (list): Dimensions of the successive feature spaces of MLP2

```

```

        with_extra (bool): Whether additional pre-computed features are
        ↪passed between the two MLPs
        extra_size (int, optional): Number of channels of the additional
        ↪features, if any.
    """
    super(PixelSetEncoder, self).__init__()

    self.input_dim = input_dim
    self.mlp1_dim = copy.deepcopy(mlp1)
    self.mlp2_dim = copy.deepcopy(mlp2)
    self.pooling = pooling

    self.with_extra = with_extra
    self.extra_size = extra_size

    self.name = 'PSE-{}-{}-{}'.format('|'.join(list(map(str, self.
    ↪mlp1_dim))), pooling,
                                                                    '|'.join(list(map(str, self.
    ↪mlp2_dim))))

    self.output_dim = input_dim * len(pooling.split('_')) if len(self.
    ↪mlp2_dim) == 0 else self.mlp2_dim[-1]

    inter_dim = self.mlp1_dim[-1] * len(pooling.split('_'))

    if self.with_extra:
        self.name += 'Extra'
        inter_dim += self.extra_size

    assert (input_dim == mlp1[0])
    assert (inter_dim == mlp2[0])
    # Feature extraction
    layers = []
    for i in range(len(self.mlp1_dim) - 1):
        layers.append(nn.Linear(self.mlp1_dim[i], self.mlp1_dim[i + 1]))
    self.mlp1 = nn.Sequential(*layers)

    # MLP after pooling
    layers = []
    for i in range(len(self.mlp2_dim) - 1):
        layers.append(nn.Linear(self.mlp2_dim[i], self.mlp2_dim[i + 1]))
        layers.append(nn.BatchNorm1d(self.mlp2_dim[i + 1]))
        if i < len(self.mlp2_dim) - 2:
            layers.append(nn.ReLU())
    self.mlp2 = nn.Sequential(*layers)

```

```

def forward(self, input):
    """
    The input of the PSE is a tuple of tensors as yielded by the
    →PixelSetData class:
        (Pixel-Set, Pixel-Mask) or ((Pixel-Set, Pixel-Mask), Extra-features)
        Pixel-Set : Batch_size x (Sequence length) x Channel x Number of pixels
        Pixel-Mask : Batch_size x (Sequence length) x Number of pixels
        Extra-features : Batch_size x (Sequence length) x Number of features

    If the input tensors have a temporal dimension, it will be combined with
    →the batch dimension so that the
        complete sequences are processed at once. Then the temporal dimension is
    →separated back to produce a tensor of
        shape Batch_size x Sequence length x Embedding dimension
    """
    a, b = input
    if len(a) == 2: # extra features included
        out, mask = a
        extra = b
        if len(extra) == 2:
            extra, bm = extra
            extra = extra.unsqueeze(1).repeat(1, out.size(1), 1).float()
        else:
            out, mask = a, b
        mask = mask.unsqueeze(1).repeat(1, out.size(1), 1)

    if len(out.shape) == 4:
        # Combine batch and temporal dimensions in case of sequential input
        reshape_needed = True
        batch, temp = out.shape[:2]

        out = out.view(batch * temp, *out.shape[2:])
        mask = mask.view(batch * temp, -1)
        if self.with_extra:
            extra = extra.view(batch * temp, -1)
        else:
            reshape_needed = False
        out = self.mlp1(out)
        out = torch.cat([pooling_methods[n](out, mask) for n in self.pooling.
    →split('_', 1)], dim=1)

    if self.with_extra:
        out = torch.cat([out, extra], dim=1)
    out = self.mlp2(out)
    if reshape_needed:
        out = out.view(batch, temp, -1)
    return out

```

```

class linlayer(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(linlayer, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim

        self.lin = nn.Linear(in_dim, out_dim)
        self.bn = nn.BatchNorm1d(out_dim)

    def forward(self, input):
        out = input.permute((0, 2, 1)) # to channel last
        out = self.lin(out)

        out = out.permute((0, 2, 1)) # to channel first
        out = self.bn(out)
        out = F.relu(out)

        return out

def masked_mean(x, mask):
    out = x.permute((1, 0, 2))
    out = out * mask
    out = out.sum(dim=-1) / mask.sum(dim=-1)
    out = out.permute((1, 0))
    return out

def masked_std(x, mask):
    m = masked_mean(x, mask)

    out = x.permute((2, 0, 1))
    out = out - m
    out = out.permute((2, 1, 0))

    out = out * mask
    d = mask.sum(dim=-1)
    d[d == 1] = 2

    out = (out ** 2).sum(dim=-1) / (d - 1)
    out = torch.sqrt(out + 10e-32) # To ensure differentiability
    out = out.permute(1, 0)
    return out

def maximum(x, mask):
    return x.max(dim=-1)[0].squeeze()

def minimum(x, mask):

```

```

        return x.min(dim=-1)[0].squeeze()

pooling_methods = {
    'mean': masked_mean,
    'std': masked_std,
    'max': maximum,
    'min': minimum
}

```

Lightweight Temporal Attention Encoder Module

```

[4]: class LTAE(nn.Module):
    def __init__(self, in_channels=128, n_head=16, d_k=8, n_neurons=[256,128],
        dropout=0.2,
            d_model=256, T=1000, len_max_seq=24, positions=None,
        return_att=False):
        """
        Sequence-to-embedding encoder.
        Args:
            in_channels (int): Number of channels of the input embeddings
            n_head (int): Number of attention heads
            d_k (int): Dimension of the key and query vectors
            n_neurons (list): Defines the dimensions of the successive feature
        →spaces of the MLP
            that processes the concatenated outputs of the attention heads
            dropout (float): dropout
            T (int): Period to use for the positional encoding
            len_max_seq (int, optional): Maximum sequence length, used to
        →pre-compute the positional
            encoding table
            positions (list, optional): List of temporal positions to use
        →instead of position
            in the sequence
            d_model (int, optional): If specified, the input tensors will first
        →processed by a
            fully connected layer to project them into a feature space of
        →dimension d_model
            return_att (bool): If true, the module returns the attention masks
        →along with the
            embeddings (default False)

        """

        super(LTAE, self).__init__()
        self.in_channels = in_channels

```

```

self.positions = positions
self.n_neurons = copy.deepcopy(n_neurons)
self.return_att = return_att

if positions is None:
    positions = len_max_seq + 1

if d_model is not None:
    self.d_model = d_model
    self.inconv = nn.Sequential(nn.Conv1d(in_channels, d_model, 1),
                                nn.LayerNorm((d_model, len_max_seq)))
else:
    self.d_model = in_channels
    self.inconv = None

sin_tab = get_sinusoid_encoding_table(positions, self.d_model // n_head,
→T=T)
self.position_enc = nn.Embedding.from_pretrained(torch.cat([sin_tab for
→_ in range(n_head)], dim=1),
                                                    freeze=True)

self.inlayernorm = nn.LayerNorm(self.in_channels)
self.outlayernorm = nn.LayerNorm(n_neurons[-1])
self.attention_heads = MultiHeadAttention(n_head=n_head, d_k=d_k,
→d_in=self.d_model)

assert (self.n_neurons[0] == self.d_model)

activation = nn.ReLU()

layers = []
for i in range(len(self.n_neurons) - 1):
    layers.extend([nn.Linear(self.n_neurons[i], self.n_neurons[i + 1]),
                    nn.BatchNorm1d(self.n_neurons[i + 1]),
                    activation])

self.mlp = nn.Sequential(*layers)
self.dropout = nn.Dropout(dropout)

def forward(self, x):

    sz_b, seq_len, d = x.shape
    x = self.inlayernorm(x)

    if self.inconv is not None:
        x = self.inconv(x.permute(0, 2, 1)).permute(0, 2, 1)

    if self.positions is None:

```

```

        src_pos = torch.arange(1, seq_len + 1, dtype=torch.long).
→expand(sz_b, seq_len).to(x.device)
        else:
            src_pos = torch.arange(0, seq_len, dtype=torch.long).expand(sz_b,
→seq_len).to(x.device)
            enc_output = x + self.position_enc(src_pos)
            enc_output, attn = self.attention_heads(enc_output, enc_output,
→enc_output)
            enc_output = enc_output.permute(1, 0, 2).contiguous().view(sz_b, -1) #
→Concatenate heads
            enc_output = self.outlayernorm(self.dropout(self.mlp(enc_output)))

        if self.return_attn:
            return enc_output, attn
        else:
            return enc_output

class MultiHeadAttention(nn.Module):
    ''' Multi-Head Attention module '''

    def __init__(self, n_head, d_k, d_in):
        super().__init__()
        self.n_head = n_head
        self.d_k = d_k
        self.d_in = d_in

        self.Q = nn.Parameter(torch.zeros((n_head, d_k))).requires_grad_(True)
        nn.init.normal_(self.Q, mean=0, std=np.sqrt(2.0 / (d_k)))

        self.fc1_k = nn.Linear(d_in, n_head * d_k)
        nn.init.normal_(self.fc1_k.weight, mean=0, std=np.sqrt(2.0 / (d_k)))

        self.attention = ScaledDotProductAttention(temperature=np.power(d_k, 0.
→5))

    def forward(self, q, k, v):
        d_k, d_in, n_head = self.d_k, self.d_in, self.n_head
        sz_b, seq_len, _ = q.size()

        q = torch.stack([self.Q for _ in range(sz_b)], dim=1).view(-1, d_k) #
→(n*b) x d_k

        k = self.fc1_k(v).view(sz_b, seq_len, n_head, d_k)
        k = k.permute(2, 0, 1, 3).contiguous().view(-1, seq_len, d_k) # (n*b) x
→lk x dk

```



```

        v = torch.stack(v.split(v.shape[-1] // n_head, dim=-1)).view(n_head *
→sz_b, seq_len, -1)
        output, attn = self.attention(q, k, v)
        attn = attn.view(n_head, sz_b, 1, seq_len)
        attn = attn.squeeze(dim=2)

        output = output.view(n_head, sz_b, 1, d_in // n_head)
        output = output.squeeze(dim=2)

        return output, attn

class ScaledDotProductAttention(nn.Module):
    ''' Scaled Dot-Product Attention '''

    def __init__(self, temperature, attn_dropout=0.1):
        super().__init__()
        self.temperature = temperature
        self.dropout = nn.Dropout(attn_dropout)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k, v):
        attn = torch.matmul(q.unsqueeze(1), k.transpose(1, 2))
        attn = attn / self.temperature

        attn = self.softmax(attn)
        attn = self.dropout(attn)
        output = torch.matmul(attn, v)

        return output, attn

def get_sinusoid_encoding_table(positions, d_hid, T=1000):
    ''' Sinusoid position encoding table
    positions: int or list of integer, if int range(positions)'''

    if isinstance(positions, int):
        positions = list(range(positions))

    def cal_angle(position, hid_idx):
        return position / np.power(T, 2 * (hid_idx // 2) / d_hid)

    def get_posi_angle_vec(position):
        return [cal_angle(position, hid_j) for hid_j in range(d_hid)]

    sinusoid_table = np.array([get_posi_angle_vec(pos_i) for pos_i in positions])

```

```

sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2]) # dim 2i
sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2]) # dim 2i+1

if torch.cuda.is_available():
    return torch.FloatTensor(sinusoid_table).cuda()
else:
    return torch.FloatTensor(sinusoid_table)

def get_sinusoid_encoding_table_var(positions, d_hid, clip=4, offset=3, T=1000):
    ''' Sinusoid position encoding table
    positions: int or list of integer, if int range(positions)'''

    if isinstance(positions, int):
        positions = list(range(positions))

    x = np.array(positions)

    def cal_angle(position, hid_idx):
        return position / np.power(T, 2 * (hid_idx + offset // 2) / d_hid)

    def get_posi_angle_vec(position):
        return [cal_angle(position, hid_j) for hid_j in range(d_hid)]

    sinusoid_table = np.array([get_posi_angle_vec(pos_i) for pos_i in positions])

    sinusoid_table = np.sin(sinusoid_table) # dim 2i
    sinusoid_table[:, clip:] = torch.zeros(sinusoid_table[:, clip:].shape)

    if torch.cuda.is_available():
        return torch.FloatTensor(sinusoid_table).cuda()
    else:
        return torch.FloatTensor(sinusoid_table)

```

Decoder

```

[5]: def get_decoder(n_neurons):
    """Returns an MLP with the layer widths specified in n_neurons.
    Every linear layer but the last one is followed by BatchNorm + ReLu

    args:
        n_neurons (list): List of int that specifies the width and length of the
        →MLP.
    """
    layers = []
    for i in range(len(n_neurons)-1):
        layers.append(nn.Linear(n_neurons[i], n_neurons[i+1]))

```

```

        if i < (len(n_neurons) - 2):
            layers.extend([
                nn.BatchNorm1d(n_neurons[i + 1]),
                nn.ReLU()
            ])
    m = nn.Sequential(*layers)
    return m

```

The final Model

```

[6]: class PseLTaeCombinedPlanetS1(nn.Module):
    """
    Pixel-Set encoder + Lightweight Temporal Attention Encoder sequence
    → classifier
    """

    def __init__(self, input_dim_planet=5, input_dim_s1=3, mlp1_planet=[5, 32,
    →64],
                    mlp1_s1=[3, 32, 64], pooling='mean_std', mlp2=[132, 128],
    →with_extra=True,
                    extra_size=4, n_head=16, d_k=8, d_model=256, mlp3_planet=[256,
    →128],
                    mlp3_s1=[256, 64], dropout=0.2, T=1000, len_max_seq_planet=244,
    →len_max_seq_s1=41, positions=None,
                    mlp4=[128+64, 64, 32, 20], return_att=False):
        super(PseLTaeCombinedPlanetS1, self).__init__()

        # if extras is true then include it only in planet model
        self.spatial_encoder_planet = PixelSetEncoder(input_dim_planet,
    →mlp1=mlp1_planet,
                                                pooling=pooling,
    →mlp2=mlp2, with_extra=with_extra,
                                                extra_size=extra_size)

        self.temporal_encoder_planet = LTAE(in_channels=mlp2[-1], n_head=n_head,
    →d_k=d_k,
                                                d_model=d_model,
    →n_neurons=mlp3_planet, dropout=dropout,
                                                T=T, len_max_seq=len_max_seq_planet,
    →positions=positions,
                                                return_att=return_att)

        mlp2_copy = mlp2.copy()
        if with_extra: mlp2_copy[0] = mlp2_copy[0] - extra_size

        self.spatial_encoder_s1 = PixelSetEncoder(input_dim_s1, mlp1=mlp1_s1,
    →pooling=pooling,

```

```

mlp2=mlp2_copy,
→with_extra=False,
                                extra_size=extra_size)
    self.temporal_encoder_s1 = LTAE(in_channels=mlp2_copy[-1],
→n_head=n_head, d_k=d_k,
                                d_model=d_model, n_neurons=mlp3_s1,
→dropout=dropout,
                                T=T, len_max_seq=len_max_seq_s1,
→positions=positions,
                                return_att=return_att)

    self.decoder = get_decoder(mlp4)
    self.return_att = return_att
    self.param_ratio()

def forward(self, input):
    """
    Args:
        input(tuple): (Pixel-Set, Pixel-Mask) or ((Pixel-Set, Pixel-Mask),
→Extra-features)
        Pixel-Set : Batch_size x Sequence length x Channel x Number of pixels
        Pixel-Mask : Batch_size x Sequence length x Number of pixels
        Extra-features : Batch_size x Sequence length x Number of features
    """

    input1, input2 = input

    out1 = self.spatial_encoder_planet(input1) # out size is 8,48,128
    out2 = self.spatial_encoder_s1(input2)

    if self.return_att:
        out1, att1 = self.temporal_encoder_planet(out1)
        out2, att2 = self.temporal_encoder_s1(out2)
        out = torch.cat([out1, out2], dim=1)
        out = self.decoder(out)
        return out, (att1, att2)
    else:
        out1 = self.temporal_encoder_planet(out1)
        out2 = self.temporal_encoder_s1(out2)
        out = torch.cat([out1, out2], dim=1)
        out = self.decoder(out)
        return out

def param_ratio(self):
    total = get_ntrainparams(self)
    s = get_ntrainparams(self.spatial_encoder_planet)
    t = get_ntrainparams(self.temporal_encoder_planet)
    s1 = get_ntrainparams(self.spatial_encoder_s1)

```

```

t1 = get_ntrainparams(self.temporal_encoder_s1)
c = get_ntrainparams(self.decoder)

print('TOTAL TRAINABLE PARAMETERS : {}'.format(total))
print('PLANET Spatial {:.1f}%'.format(s / total * 100))
print('PLANET Temporal {:.1f}%'.format(t / total * 100))
print('SENTINEL-1 Spatial {:.1f}%'.format(s1 / total * 100))
print('SENTINEL-1 Temporal {:.1f}%'.format(t1 / total * 100))
print('Classifier {:.1f}%'.format(c / total * 100))

return total

```

1.4.1 Hyperparameters

The hyperparameter optimization was included by us using NNI (<https://nni.readthedocs.io/en/stable/>). However, even with extensive hyperparameter optimization, the best results were achieved using the default network configuration.

- General parameters
 - batch_size: 8
 - with_extra: False (i.e., do not include crop_area and crop_len)
 - augmentation: False (no Gaussian random noise during training)
- Pixel-Set Encoder module
 - mlp1-in: 32, mlp1-out: 64 (Dimensions of the successive feature spaces of MLP1, Planet)
 - mlp1-s1-in: 32, mlp1-s1-out: 64 (Dimensions of the successive feature spaces of MLP1, Sentinel-1)
 - factor: 16 (Scale for MLP3 input dimension)
 - mlp3-out: 128 (MLP3 output dimension, Planet)
 - mlp3-s1-out: 128 (MLP3 output dimension, Sentinel-1)
- Temporal attention module
 - n-head: 16 (attention heads)
 - d-k: 8 (in temporal attention module)
 - dropout: 0.2
- Decoder module
 - mlp4-1: 64, mlp4-2: 64 (Dimensions of the successive feature spaces of MLP4)
 - scale: 0.25 (sentinel-1 data weighted relative to planet data in the decoder layer of the combined model)

1.4.2 Training Details

The given dataset for training and validation was augmented as described in the preprocessing step, creating 10 samples out of each image. These samples were then randomly divided in training and validation set, however keeping (augmented) samples from the same field in either the training or validation set, in order to prevent data leakage. In total 90% of the data was used for training and 10% was kept for validation in each fold of the cross-validation.

We use the Adam optimizer with the learning rate 0.001 and weight decay 0.000001. The training includes the monitoring of several metrics, which are calculated in

./ai4food/evaluation_utils.py. Accuracy was used as the early stopping condition with a patience of 10 epochs.

As loss function, focal loss is used, which is not exactly equal to the evaluation metric the model is finally evaluated on, but is close enough for efficient training and is also used in Garnot. et al. (2020). The script is also copied to the Dockerimage. During training, both the training and the validation loss are close to zero, which indicates overfitting. This is countered with a k -fold validation scheme.

```
[7]: """
Credits to github.com/clcarwin/focal\_loss\_pytorch
"""

import torch
import torch.nn.functional as F
from torch.autograd import Variable
import torch.nn as nn

class FocalLoss(nn.Module):
    def __init__(self, gamma=0, alpha=None, size_average=True):
        super(FocalLoss, self).__init__()
        self.gamma = gamma
        self.alpha = alpha
        if isinstance(alpha, (float, int)): self.alpha = torch.Tensor([alpha, 1-
→ alpha])
        if isinstance(alpha, list): self.alpha = torch.Tensor(alpha)
        self.size_average = size_average

    def forward(self, input, target):
        if input.dim() > 2:
            input = input.view(input.size(0), input.size(1), -1) # N,C,H,W =>N, C, H*W
            input = input.transpose(1, 2) # N,C,H*W => N,H*W,C
            input = input.contiguous().view(-1, input.size(2)) # N,H*W,C =>N*H*W, C
            target = target.view(-1, 1)

        logpt = F.log_softmax(input, dim=1)
        logpt = logpt.gather(1, target)
        logpt = logpt.view(-1)
        pt = Variable(logpt.data.exp())

        if self.alpha is not None:
            if self.alpha.type() != input.data.type():
                self.alpha = self.alpha.type_as(input.data)
            at = self.alpha.gather(0, target.data.view(-1))
```

```

        logpt = logpt * Variable(at)

    loss = -1 * (1 - pt) ** self.gamma * logpt
    if self.size_average:
        return loss.mean()
    else:
        return loss.sum()

```

For the loss computation, we set alpha as the inverse of the class frequencies in the training data, which were found during the exploration phase. In this way we aim to balance the class distribution.

The training routing can be found in `./ai4food/training.py`. To reproduce our model training, execute

```

python3 training.py --num-workers 0 --input-data planet sentinel-1 --split
train --ndvi 1 --nri 1 --input-dim 5 3 --include-extras 0 --dev-data-dir
/work/shared_data/2021-ai4food/dev_data/south-africa --nr-classes 5
--input-data-type extracted-640 --k-fold 10

```

Each model of the 10-fold cross validation, may be trained for a maximum of 100 epochs, but early stopping is applied, which stops the training routine after 10 epochs if no improvement to the validation accuracy is observed. Then the best performing model in terms of validation accuracy is saved.

To run inference with the trained models on the test set and generate a submission file, execute

```

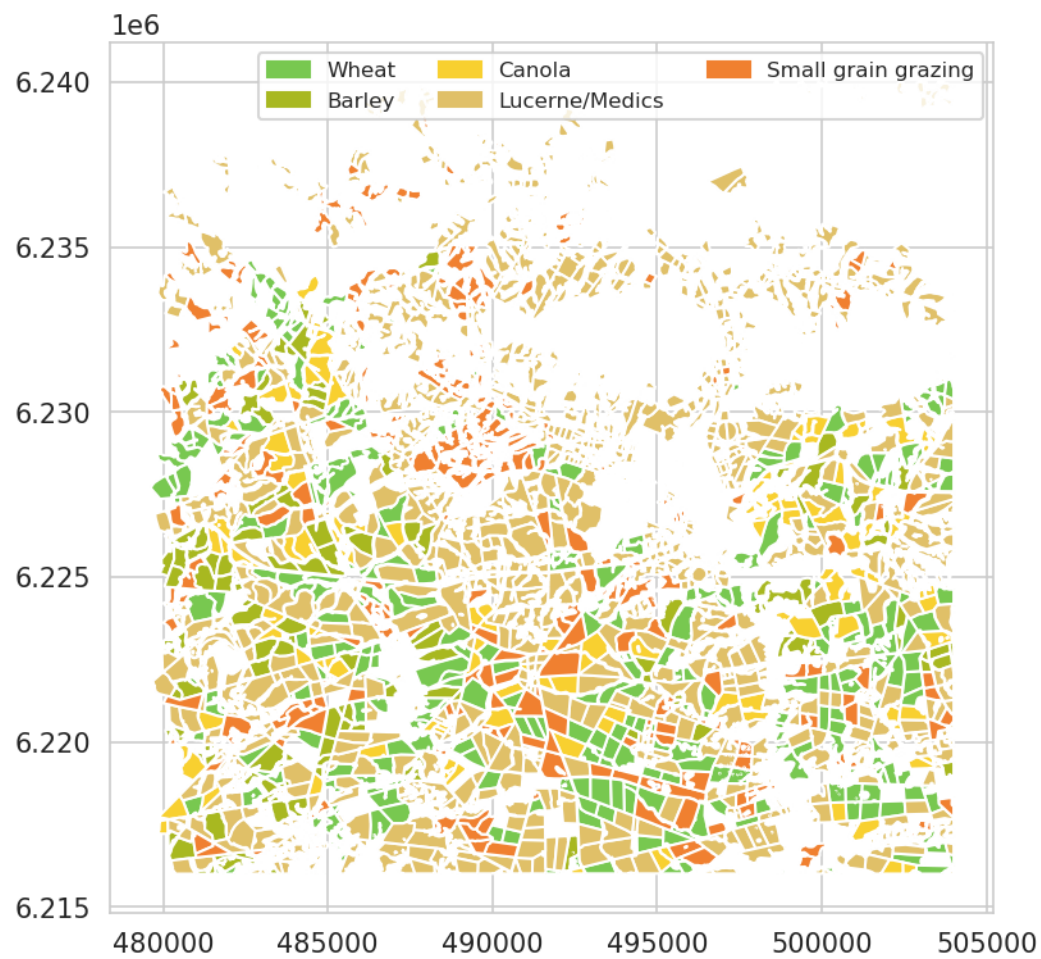
python3 training.py --num-workers 0 --input-data planet sentinel-1 --split
test --ndvi 1 --nri 1 --input-dim 5 3 --include-extras 0 --save-preds
--dev-data-dir /work/shared_data/2021-ai4food/dev_data/south-africa --nr-classes 5
--input-data-type extracted-640 --k-fold 10

```

During inference all 10 models of the cross validation are used and an average of the probabilities of each model is obtained to get the final prediction for each sample.

1.5 Results

Crop fields colored with the predicted crop ID, as obtained from our best submission with a score of 3.60



[]: