ISSonDL2021 Challenge

2° place solution

Colomba Luca

Outline

- Introduction and Problem Definition
- Software and Frameworks
- Dataset: Train and Validation
- Data Augmentation and Normalization
- 1° round – ResNet model
- 2° round – Models comparison
- Code

Introduction and Problem Definition

- Problem: Waste classification
- Task: Binary classification task on images (Organic vs. Recyclable)

[redacted]

Organic sample image

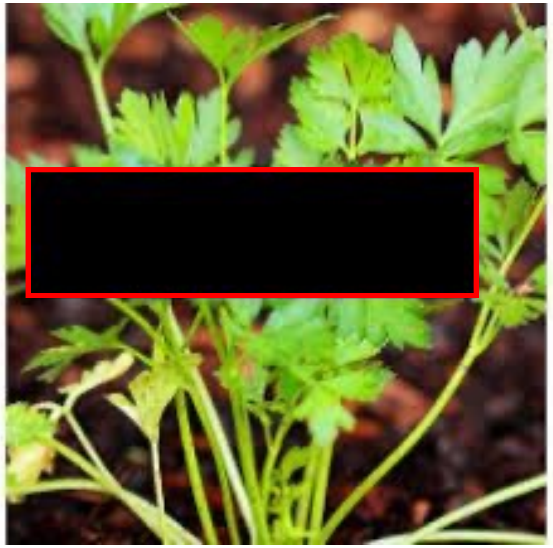
[redacted]

Recyclable sample image

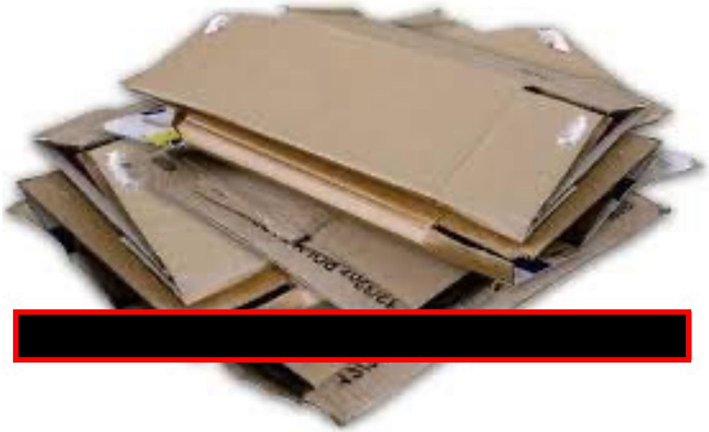
Introduction and Problem Definition

Issue: images were covered with black rectangles.

I did not insert any specific pre-processing operation to take into account such black rectangles, but indeed it would be interesting to analyse the impact of some pre-processing operations on the final result to furtherly improve the solution.



Organic sample image



Recyclable sample image

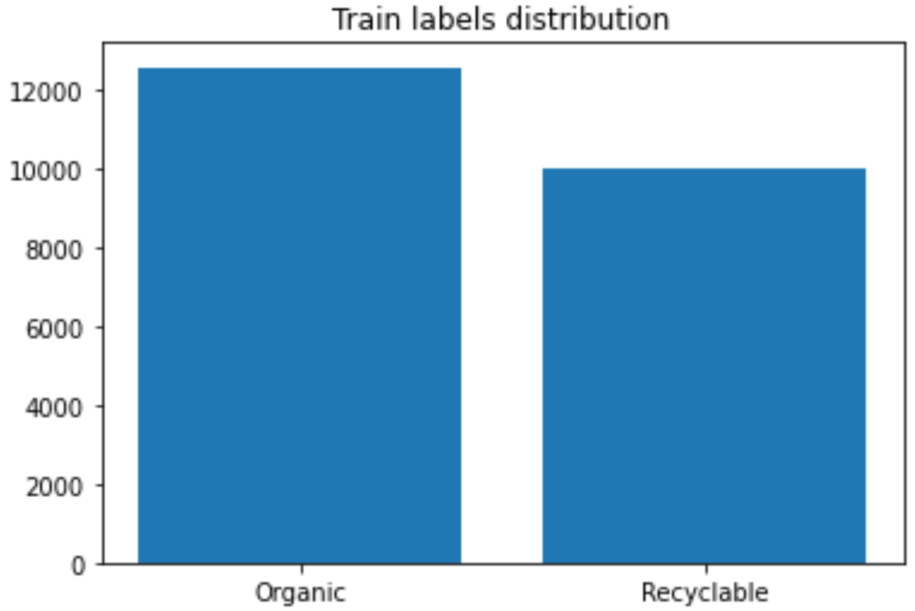
Software and Frameworks

The neural networks (NNs) were trained on Google Colab, using Python, PyTorch and PyTorch Lightning frameworks.

PyTorch Lightning is a framework developed to reduce boilerplate code when developing NNs in PyTorch, simplifying the way we write train and validation loops.

Dataset: Train and Validation

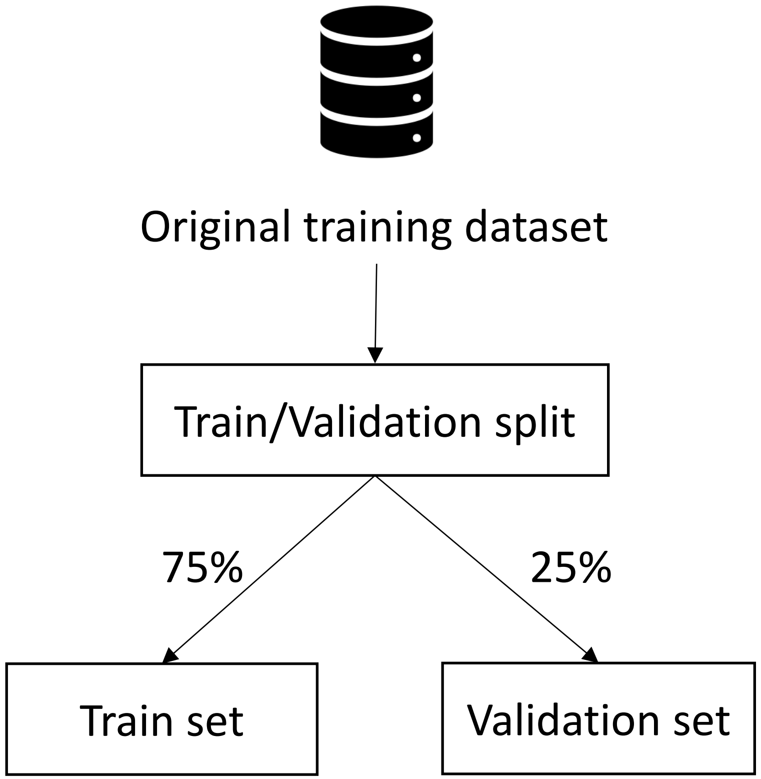
The **training** dataset provided for the challenge is composed of 22.564 images: 12.565 of class ‘Organic’ and 9.999 of class ‘Recyclable’.



Dataset: Train and Validation

To validate the performances of the trained models, the training dataset was split into a train and validation set with a 75/25 split: 75% for the training data, 25% for the validation data.

Organic and Recyclable images were equally split among the two.



Original training dataset

Train/Validation split

75%

25%

Train set

Validation set

Data Augmentation and Normalization

During the training process, the following data augmentation techniques were used:

Data Augmentation	Parameters
Random Rotation	Range [-30°, +30°]
Random Horizontal Flip	Probability of 50%
Random Vertical Flip	Probability of 50%
Random Autocontrast	Probability of 50%

For additional information, you can check PyTorch documentation: https://pytorch.org/vision/stable/transforms.html

Data Augmentation and Normalization

All images were resized to a resolution of 250x250 pixels and were normalized in range [-1, +1] by subtracting a mean of 0.5 and by dividing for 0.5.

$$p_{new} = \frac{p - 0.5}{0.5}$$

Where *p* is the value of each individual pixel.

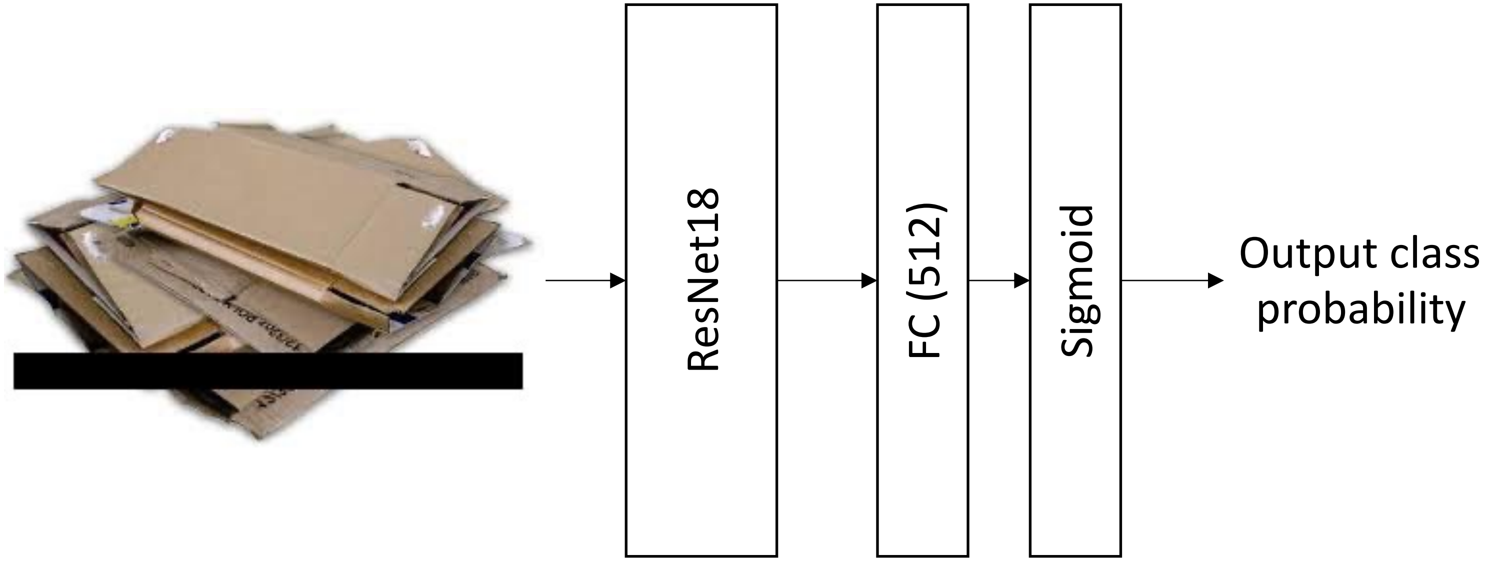
1° round – ResNet model

- Since the objects represented in the dataset are common objects, we can use pre-trained models as a starting point.
- Some of these are: ResNet, ResNeXt, DenseNet, VGG

1° round – ResNet model

For the first round, a pre-trained ResNet18 model was chosen.

The final fully connected (FC) layer was changed to perform a binary classification. The last layer has an input dimension of 512 and an output dimension of 1, with a Sigmoid non-linear function.


[redacted]

1° round – ResNet model

Due to the limited amount of time and resources for the GPU instance given by Google Colab, the training was limited to **10 epochs** with **Binary Cross Entropy l**oss function.

The same parameters and optimizer were used in Round 2.

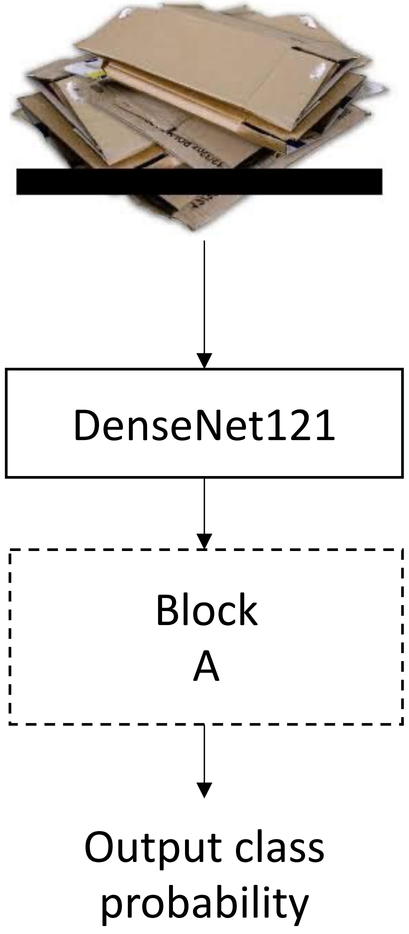
The model achieved an accuracy of **91.26%** on the validation set.

Loss function	Binary Cross Entropy
Epochs	10
Batch size	64
Optimizer	Adam
Learning Rate	$1 \cdot 10^{-3}$
Weight Decay	0

2° round – DenseNet model

In the second round, DenseNet121 model was considered. More specifically, 4 different variations with different final layer (Block A):

		DenseNet Simple	DenseNet_128	DenseNet_256	DenseNet_512
FC Layer1	Input Size	1024	1024	1024	1024
	Output Size	1	128 + LeakyReLU (+ Dropout 20%)	256 + LeakyReLU (+ Dropout 20%)	512 + LeakyReLU (+ Dropout 20%)
FC Layer2	Input Size	N/A	128	256	512
	Output Size	N/A	1	1	1
Nonlinear Function		Sigmoid	Sigmoid	Sigmoid	Sigmoid


[redacted]

DenseNet121

Block
A

Output class probability

2° round – DenseNet model

The following accuracies were achieved on the validation set:

Model	Accuracy
ResNet (Round 1)	91.26%
DenseNet Simple	92.71%
DenseNet_128	94.20%
DenseNet_256	94.18%
DenseNet_512	**94.65%**

Code

You can find the repository containing my solution at the following link:

https://github.com/lccol/issondl2021-challenge

In the following slides, the main aspects of the code are reported.

Code - Imports

```python
import torch
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import cv2
import pytorch_lightning as pl
import pickle as pkl
from pathlib import Path
from torch import nn, optim
from torch import functional as F
from torchvision import models
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
from functools import partial
from PIL import Image
from collections import defaultdict
```

Code – Train and Validation split

```
def split(*lists, seed=47, test_perc=.25):
    r = random.Random(seed)
    res = []
    for l in lists:
        r.shuffle(l)
        test_size = int(len(l) * test_perc)
        train_size = len(l) - test_size
        train, test = l[:train_size], l[-test_size:]
        assert len(l) == len(train) + len(test)
        res.append((train, test))
    return res
organic_files = [str(x) for x in (train_path / organic_foldername).iterdir()
                           if x.is_file()]
recyclable_files = [str(x) for x in (train_path /
                           recyclable_foldername).iterdir() if x.is_file()]

organic_tuple, recyclable_tuple = split(organic_files, recyclable_files,
                                        seed=47, test_perc=.25)
organic_train, organic_validation = organic_tuple
recyclable_train, recyclable_validation = recyclable_tuple
```

Code – Custom Dataset class for Test set

```
class ImageLoader(Dataset):
    def __init__(self, path, transform):
        self.path = path
        self.transform = transform
        self.all_images = [x for x in os.listdir(path) \
                            if os.path.isfile(os.path.join(path, x))]

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

    def __getitem__(self, idx):
        path = os.path.join(self.path, self.all_images[idx])
        image = Image.open(path).convert('RGB')
        tensor_image = self.transform(image) if self.transform else image

        filename = os.path.splitext(self.all_images[idx])[0]
        return filename, tensor_image
```

Code – PyTorch Transformations

```
train_transformers = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomAutocontrast(),
    transforms.Resize((250, 250)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,) * 3, (0.5,) * 3)
])

test_transformers = transforms.Compose([
    transforms.Resize((250, 250)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,) * 3, (0.5,) * 3)
])
```

Transforms are PyTorch’s common image transformations that are applied to each image. They are useful for normalization and data augmentation steps.

Code – Utilities

```
def file_validator(filepath, checklist):
        return filepath in checklist

def get_files(*lists):
    # Generate a single list containing all the elements
    starter = lists[0].copy()
    for l in lists[1:]:
    starter.extend(l)
    return starter


train_list = get_files(organic_train, recyclable_train)
validation_list = get_files(organic_validation, recyclable_validation)

assert len(set(train_list).intersection(set(validation_list))) == 0

train_validator = partial(file_validator, checklist=set(train_list))
validation_validator = partial(file_validator, checklist=set(validation_list))
```

Generate train and validation lists: two lists containing all the filenames used for training and validation sets and verify that there is no intersection.

The validators are used in the next slide as argument to ImageFolder class to correctly load only the necessary images.

Code – Datasets creation

```
train_dataset = ImageFolder(train_path, transform=train_transformers,
                           is_valid_file=train_validator)
validation_dataset = ImageFolder(train_path, transform=test_transformers,
                           is_valid_file=validation_validator)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,
                           drop_last=False, num_workers=2)
validation_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE,
                           shuffle=True, drop_last=False, num_workers=2)

assert len(train_dataset) == len(train_list)
assert len(validation_list) == len(validation_dataset)

test_dataset = ImageLoader(test_path, test_transformers)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=1,
                           drop_last=False, num_workers=2)
```

Code – PyTorch Lightning Module (1/3)

```
class LightningWrapper(pl.LightningModule):
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.loss = nn.BCEWithLogitsLoss()
        self.val_numerator = 0
        self.val_denominator = 0
        self.mylogs = defaultdict(list)
        self.loss_accumulator = 0
        self.batch_counter = 0

    def forward(self, x):
        pred = self.model(x)
        return pred
    def backward(self, loss, optimizer, optimizer_idx):
        loss.backward()

    def configure_optimizers(self):
        optimizer = optim.Adam(self.model.parameters(), lr=1e-3)
        return optimizer
```

Code – PyTorch Lightning Module (2/3)

```
def training_step(self, train_batch, batch_idx):
    x, y = train_batch
    y = y.float()
    pred = self.model(x).squeeze()
    loss = self.loss(pred, y)
    self.log('train_loss_per_step', loss)
    self.mylogs['train_loss_per_step'].append(loss)

    self.loss_accumulator += loss
    self.batch_counter += 1
    return loss

def training_epoch_end(self, training_step_outputs):
    self.mylogs['my_train_loss_avg'].append(self.loss_accumulator /
                                            self.batch_counter)
    self.batch_counter = 0
    self.loss_accumulator = 0
    return
```

Code – PyTorch Lightning Module (3/3)

```
def validation_step(self, validation_batch, batch_idx):
    x, y = validation_batch
    y = y.float()
    pred = self.model(x).squeeze()

    pred_npy = (torch.sigmoid(pred) >= 0.5).cpu().numpy()
    y_npy = y.cpu().numpy()
    self.val_numerator += (pred_npy == y_npy).sum()
    self.val_denominator += y_npy.size
    loss = self.loss(pred, y)
    self.log('validation_loss', loss)
    self.mylogs['validation_loss'].append(loss)
    return

def validation_step_end(self, batch_parts):
    accuracy = self.val_numerator / self.val_denominator
    self.val_numerator = 0
    self.val_denominator = 0
    self.log('validation_accuracy', accuracy)
    self.mylogs['validation_accuracy'].append(accuracy)
    return
```

Code – PyTorch Models

```
net = models.resnet18(pretrained=True)
for p in net.parameters():
       p.requires_grad = False
net.fc = nn.Linear(512, 1)
```

ResNet Model

```
net = models.densenet121(pretrained=True)
for p in net.parameters():
       p.requires_grad = False
net.classifier = nn.Linear(1024, 1)
```

DenseNet Simple Model

```
net = models.densenet121(pretrained=True)
for p in net.parameters():
       p.requires_grad = False
net.classifier = nn.Sequential(
    nn.Linear(1024, X),
    nn.LeakyReLU(),
    nn.Dropout(0.2),
    nn.Linear(X, 1)
)
```

DenseNet_X Model

Code – PyTorch Lightning Training

```
pl_net = LightningWrapper(net)
trainer = pl.Trainer(max_epochs=10, check_val_every_n_epoch=1, gpus=1)
trainer.fit(pl_net, train_loader, validation_loader)
torch.save(pl_net.model, ‘…')

logger_info = dict(pl_net.mylogs)
with open(‘…’, 'wb') as fp:
       pkl.dump(logger_info, fp)
```

Code – Retrain on entire set for submission

```
# TRAIN ON THE ENTIRE DATASET
all_train_dataset = ImageFolder(train_path, transform=train_transformers)
all_train_loader = DataLoader(all_train_dataset, batch_size=BATCH_SIZE,
                                 shuffle=True, drop_last=False)

net_all = models.densenet121(pretrained=True)
for p in net_all.parameters():
        p.requires_grad = False
net_all.classifier = nn.Sequential(
    nn.Linear(1024, 512),
    nn.LeakyReLU(),
    nn.Dropout(0.2),
    nn.Linear(512, 1)
)
pl_net_all = LightningWrapper(net_all)
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(pl_net_all, all_train_loader)
torch.save(pl_net_all.model, ‘…')
```

Code – Submission CSV generation (1/2)

```
test_transformers = transforms.Compose([
    transforms.Resize((250, 250)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,) * 3, (0.5,) * 3)
])

test_path2 = Path('/content', 'dataset_stage_2')
test_dataset = ImageLoader(test_path2, test_transformers)

res_df = defaultdict(list)
# 'id', 'label'
res_df_export_path = ‘…'

final_model = pl_net_all.model.to(device)
class_to_idx = all_train_dataset.class_to_idx
print(f'class_to_idx: {class_to_idx}')
conversion_dict = {class_to_idx[k]: k for k in class_to_idx}
print(f'conversion dict: {conversion_dict}')
```

Code – Submission CSV generation (2/2)

```
with torch.no_grad():
    final_model.eval()
    for idx in range(len(test_dataset)):
        identifier, x = test_dataset[idx]
        x = x.unsqueeze(0).to(device)

        pred = final_model(x)
        pred_sigm = torch.sigmoid(pred)
        pred_class = (pred_sigm >= 0.5).squeeze(0).cpu().numpy()
        assert len(pred_class) == 1
        pred_class = pred_class[0]
        if pred_class:
                pred_class = 1
        else:
                pred_class = 0

        pred_class_str = conversion_dict[pred_class]
        res_df['id'].append(identifier)
        res_df['label'].append(pred_class_str)

df = pd.DataFrame(res_df)
df.to_csv(res_df_export_path, index=False)
```