Last but not list - Parte 2

July 8, 2023

Corso di Introduzione al Machine Learning, A.A. 2022-2023

Gruppo *Last but not list* - Elio Grande, Francesco Zigliotto

1 Introduzione

Scopo del presente lavoro è stato sperimentare tecniche avanzate di *Machine Learning*, accludendo un po' di analisi dei dati. Tra i dataset proposti abbiamo optato per il *New Plant Disease Dataset*, disponibile alla pagina web https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset (ultimo accesso 04/07/2023). Si tratta di fotografie - in presa da laboratorio e non sul campo - di foglie appartenenti a piante di diverse specie, talvolta sane e talaltra affette da certe malattie (secondo un massimo di una malattia per foglia). Il dataset è detto *augmented*: i soggetti fotografici hanno certe peculiarità (come l'essere ritratti in posizioni varie) atte a rendere più efficace l'allenamento di un classificatore eventualmente da utilizzare *in the wild*. Nello specifico, il nostro compito è consistito nel costruire un *Neural Network* capace di riconoscere sia la specie che la malattia della fogliolina ritratta. Il modello adoperato, più avanti descritto nei particolari e ispirato alla bibliografia indicata nella sezione corrispondente di questo report - è un *Convolutional Neural Network* in stile tradizionale, ovvero costituito da un blocco convoluzionale ed un blocco lineare *fully connected*. I risultati sono infine consultabili nella sezione "Conclusioni". Dopo aver proceduto dapprima alla definizione di alcuni strumenti utili, abbiamo importato il dataset ed estratto alcune informazioni sulla sua struttura, anche tramite visualizzazione. Dopodiché abbiamo definito le funzioni di *training* e validazione del modello, subito dopo costruito sotto forma di class-object ed istanziato. Ha fatto seguito l'allenamento vero e proprio, avendo cura di selezionare gli iperparametri migliori. Una fase di test - in verità una doppia fase data l'esigua dimensione del test set fornito, che ci ha costretti a ricavare un secondo test set apprezzabilmente esteso - ha infine ultimato il nostro lavoro, coronata da alcune valutazioni.

Una nota sull'analisi preparatoria dei dati ed i suoi limiti: i dati su cui operare sono qui di natura assai diversa dai record tabulari finora trattati. Infatti, in un record tabulare ogni istanza è un collettore di attributi, dove ciascun attributo è una variabile che pesca valori entro un dominio. Al contrario, un'immagine è sotto un certo rispetto un insieme di valori contenente i propri attributi. Un'immagine è difatti composta da un supporto materiale (qui: dei pixel tradotti in numeri compresi tra 0 e 1 inquadrati in una matrice, riproducibili sotto forma di segnali luminosi a video), che astrattamente "contiene" l'oggetto rappresentato insieme alle sue qualità (appunto, la forma arrotondata ed il colore verdastro di una fogliolina malata). Una porzione di tali attributi è nota fin da principio: sono le informazioni direttamente o indirettamente fornite insieme al dataset (i nomi della specie di appartenenza, il numero di casi etc.), e su queste abbiamo potuto indagare ad agio in fase preparatoria. Indagare su alcune altre proprietà dei dati, invece, avrebbe per lo

meno posto difficoltà oltre le nostre competenze. Si consideri ad esempio l'individuazione di *missing values*: difficilmente un'immagine presenta valori propriamente mancanti, tutt'al più valori fuori norma. Infine, svelare i contenuti dei nostri dati, ovvero in definitiva riconoscere l'oggetto raffigurato, costituiva il fine del nostro lavoro e non la fase iniziale.

1.1 Definizione degli strumenti

1.1.1 Importazione dei pacchetti necessari

[1]:
```
import numpy as np
import os
import matplotlib.pyplot as plt
import torch
import torchvision as tv
from torch.utils.data import DataLoader
from collections import Counter
from timeit import default_timer as timer
from ipywidgets import FloatProgress
from IPython.display import display
import random
from PIL import Image
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
import datetime
```

1.1.2 Specifiche del device

[2]:
```
if torch.backends.mps.is_available() and torch.backends.mps.is_built():
    device = torch.device('mps')
    print('Using MPS device')
else:
    device = torch.device('cpu')
    print('Using CPU device')
```

```
Using MPS device
```

1.1.3 Definizione preliminare di alcune funzioni utili

Definiamo la funzione `heatmap` come nel nostro precedente progetto:

[3]:
```
def heatmap(x, y, z, ax=None, xrot=0, yrot=0, vmin=-1, vmax=1):
    if ax == None:
        ax = plt.gca()
    ax.imshow(np.transpose(z), vmin=vmin, vmax=vmax)

    # Labels
    ax.set_xticks(np.arange(len(x)), labels=x)
```

```
    ax.set_yticks(np.arange(len(y)), labels=y)
    plt.setp(ax.get_xticklabels(), rotation=xrot,
             ha="right", rotation_mode="anchor")
    plt.setp(ax.get_yticklabels(), rotation=yrot,
             ha="right", rotation_mode="anchor")

    # White numeric marker
    for i in range(len(x)):
        for j in range(len(y)):
            if z[i, j] > 0:
                ax.text(i, j, '{:.0f}'.format(z[i, j]),
                        ha="center", va="center", color="w")
```

[4]:
```
def print_image(tensor, ax):
    ax.axis('off')
    ax.imshow(tensor.permute(1,2,0))
```

2 Importazione del dataset

2.1 Informazioni preliminari

Non presente di default nelle librerie *Torch* e *tv*, il *New Plant Disease Dataset (Augmented)* dev'essere importato, dopo il download manuale secondo le istruzioni fornite, da una directory presente sul disco. Le principali informazioni relative alle caratteristiche del dataset sono contenute nei nomi della directory stessa (cartelle, sottocartelle e file). Il dataset appare inoltre suddiviso in un *training set* e un *validation set*:

[5]:
```
DIR = './archive/New Plant Diseases Dataset(Augmented)/New Plant Diseases␣
 ↪Dataset(Augmented)/'
next(os.walk(DIR))[1]
```

[5]:
```
['valid', 'train']
```

a cui si aggiunge un piccolo *test set*, di alcuni file del quale, unicamente a titolo di esempio, riportiamo il titolo:

[6]:
```
test_dir = next(os.walk('./archive/test/test'))[2]
test_dir[:10]
```

[6]:
```
['AppleScab3.JPG',
 'TomatoEarlyBlight2.JPG',
 'TomatoEarlyBlight3.JPG',
 'PotatoHealthy1.JPG',
 'AppleScab2.JPG',
 'TomatoEarlyBlight1.JPG',
 'PotatoHealthy2.JPG',
 'AppleScab1.JPG',
```

```
 'TomatoEarlyBlight4.JPG',
 'TomatoEarlyBlight5.JPG']
```

Assumendo l'uniformità, ovvero che tutte le immagini originali abbiano le medesime dimensioni, l'osservazione di un singolo esempio ci consente di individuare tale formato in 256×256 pixel:

[7]:
```
Image.open('./archive/test/test/AppleScab3.JPG').size
```

[7]:
```
(256, 256)
```

Importiamo adesso il dataset nei suoi tre componenti appena menzionati. Nell'importazione riduciamo la dimensione delle immagini a 64×64 pixel. Infatti, durante la fase di training della rete neurale abbiamo osservato come laborare su immagini a dimensione ridotta renda sostanzialmente più rapida l'esecuzione del training senza comprometterne particolarmente i risultati. Dopo vari tentativi, abbiamo individuato tale dimensione come ottimale.

[8]:
```
resize = tv.transforms.Compose(
    [tv.transforms.Resize(64), tv.transforms.ToTensor()]
)

original_train_set = tv.datasets.ImageFolder(root = DIR + 'train',
                                             transform = resize)
original_valid_set = tv.datasets.ImageFolder(root = DIR + 'valid',
                                             transform = resize)
minitest_set = tv.datasets.ImageFolder(root = './archive/test',
                                       transform = resize)

print(f'Train set: {len(original_train_set)} images')
print(f'Valid set: {len(original_valid_set)} images')
print(f'Test set: {len(minitest_set)} images')
```

```
Train set: 70295 images
Valid set: 17572 images
Test set: 33 images
```

Nel caso di `train_set` e `valid_set`, il *label* di ciascuna immagine è indicato nel nome della sottocartella nella forma `[specie]___[malattia]` e viene automaticamente associato a ogni immagine in forma numerica. La lista `classes`, che conta 38 possibili etichette, associa ad ogni valore numerico il nome originale del label.

[9]:
```
print(f'train_set.classes == valid_set.classes: \
      {original_train_set.classes == original_valid_set.classes}')
classes = original_train_set.classes
print(f'Esempio: classes[0]: {classes[0]}')
print(f'Numero etichette: {len(classes)}')
```

```
train_set.classes == valid_set.classes:       True
Esempio: classes[0]: Apple___Apple_scab
Numero etichette: 38
```

Nel caso del `minitest_set`, invece, come abbiamo visto, il label dell'immagine è contenuto, in un formato diverso da prima, nel nome del file. Pertanto i label del `minitest_set` vanno scritti manualmente:

[10]:
```
minitest_labels = [('Apple', 'Cedar_apple_rust')] * 4 \
    + [('Apple', 'Apple_scab')] * 3 \
    + [('Corn_(maize)', 'Common_rust_')] * 3 \
    + [('Potato', 'Early_blight')] * 5 \
    + [('Potato', 'healthy')] * 2 \
    + [('Tomato', 'Early_blight')] * 6 \
    + [('Tomato', 'healthy')] * 4 \
    + [('Tomato', 'Tomato_Yellow_Leaf_Curl_Virus')] * 6
```

Osserviamo tuttavia che il numero di immagini presenti nel `minitest_set` è troppo piccolo per fornire una stima adeguata dell'accuratezza finale del modello. Questo set di 33 immagini sembra più pensato per testare il modello manualmente su un'immagine alla volta. Pertanto abbiamo convenuto di creare un nuovo *test set* togliendo una porzione di immagini dal *train set*:

[11]:
```
torch.manual_seed(0)

train_size = int(0.75 * len(original_train_set))
test_size = len(original_train_set) - train_size
train_set, test_set = torch.utils.data.random_split(original_train_set,
                                                    [train_size, test_size])
valid_set = original_valid_set

print(f'Train set: {len(train_set)} images')
print(f'Valid set: {len(valid_set)} images')
print(f'Test set: {len(test_set)} images')
print(f'Mini test set: {len(minitest_set)} images')
```

```
Train set: 52721 images
Valid set: 17572 images
Test set: 17574 images
Mini test set: 33 images
```

In questo modo, ora la suddivisione dei samples tra *train set*, *valid set* e *test set* è, rispettivamente 60%, 20% e 20%. Siamo pronti per preparare i *data loader*. Scegliamo il `BATCH_SIZE` di 32, in conformità con quanto fatto in [1] (v. bibliografia alla fine).

[12]:
```
torch.manual_seed(0)

BATCH_SIZE = 32

train_dataloader = DataLoader(train_set,
    batch_size = BATCH_SIZE,
    shuffle = True
)
```

```
valid_dataloader = DataLoader(valid_set,
    batch_size = BATCH_SIZE,
    shuffle = False
)

test_dataloader = DataLoader(test_set,
    batch_size = BATCH_SIZE,
    shuffle = False
)
```

2.2 Esplorazione del dataset

Ogni elemento dei dataset ottenuti (a parte `minitest_set`) è una coppia (X, y) dove X contiene l'immagine RGB in forma tensoriale $3 \times 64 \times 64$ e y è il valore numerico dell'etichetta:

[13]:
```
print('X:', train_set[0][0].shape)
print('Esempio label (y):', train_set[0][1], '->', classes[train_set[0][1]])
```

```
X: torch.Size([3, 64, 64])
Esempio label (y): 2 -> Apple___Cedar_apple_rust
```

I valori dei pixel sono nell'intervallo $[0, 1]$ (e non $[0, 255]$):

[14]:
```
torch.manual_seed(0)
images, _ = next(iter(train_dataloader)) # We take the first BATCH as sample
print ('I valori dei pixel nel training_set vanno da {:.2} a {:.2}.'
       .format(torch.min(images), torch.max(images)))

images_v, _v = next(iter(valid_dataloader)) # We take the first BATCH as sample
print ('I valori dei pixel nel valid_set vanno da {:.2} a {:.2}.'
       .format(torch.min(images_v), torch.max(images_v)))
```

```
I valori dei pixel nel training_set vanno da 0.0 a 1.0.
I valori dei pixel nel valid_set vanno da 0.0 a 1.0.
```

[15]:
```
print(images.shape)
```

```
torch.Size([32, 3, 64, 64])
```

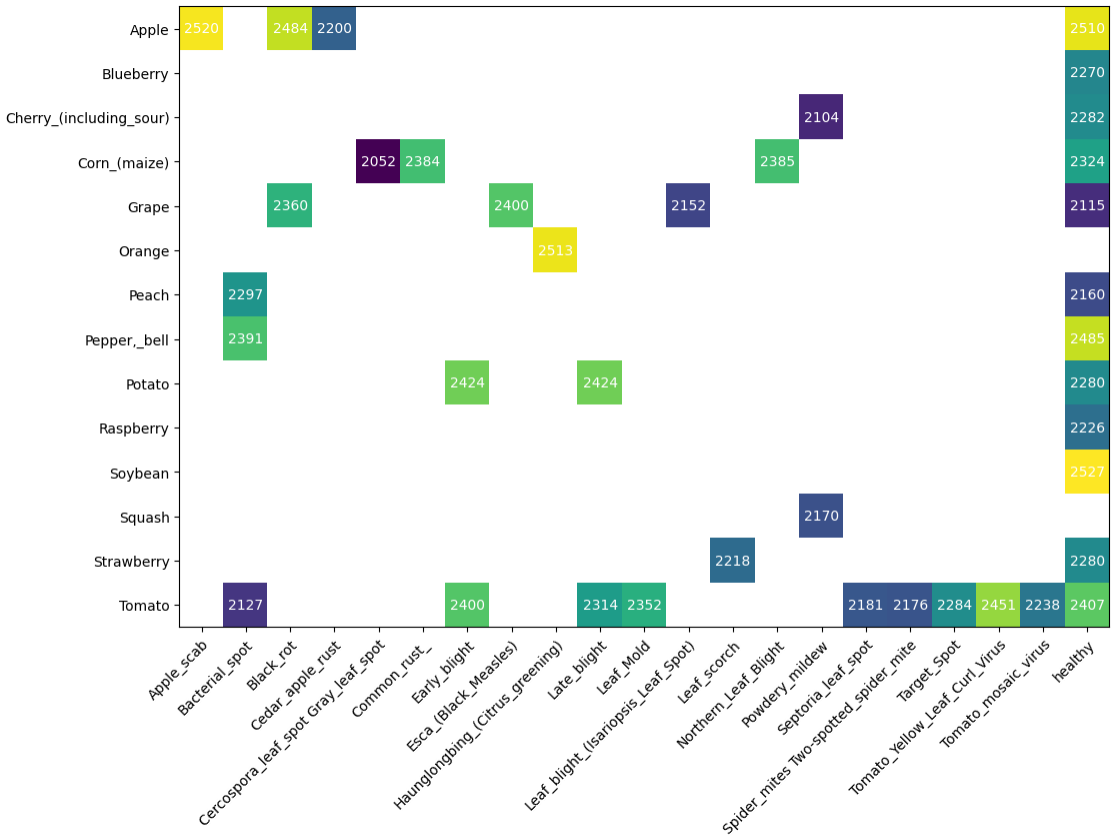
Per analizzare il nostro dataset, prepariamo una lista di label nella forma `('specie', 'malattia')` e creiamo una struttura (`dict` of `tuple`) che associ ad ogni specie e malattia il numero di immagini corrispondenti nell'intero dataset:

[16]:
```
# Count occurrences of each label
popolation_num = dict(Counter(original_train_set.targets) +␣
 ↪Counter(original_valid_set.targets))

# Trasform the list of labels into a coupled format
def parse_class(s):
```

```
    return tuple(s.split('___'))

parsed_classes = [parse_class(c) for c in classes]

# Count occurrences of each label in the form (specie, disease)
population = {}
for (specie, disease) in parsed_classes:
    population[(specie, disease)] = (len(os.listdir(DIR + 'train/'+ specie +␣
 ↪'___' + disease)) + len(os.listdir(DIR + 'valid/' + specie + '___' +␣
 ↪disease)))

# List of unique species and diseases
unique_species = sorted(list(set([l[0] for l in parsed_classes])))
unique_diseases = sorted(list(set([l[1] for l in parsed_classes])))
```

Visualizziamo ora il numero di occorrenze per ogni `('specie', 'malattia')` in una heatmap:

[17]:
```
zs = np.zeros((len(unique_diseases), len(unique_species)))

for i, x in enumerate(unique_diseases):
    for j, y in enumerate(unique_species):
        if (y,x) not in population:
            zs[i][j] = float('nan')
        else:
            zs[i][j] = population[(y,x)]

plt.figure(figsize = (12,12))
heatmap(unique_diseases, unique_species, zs, xrot=45,
        vmin=np.nanmin(zs), vmax=np.nanmax(zs))
```



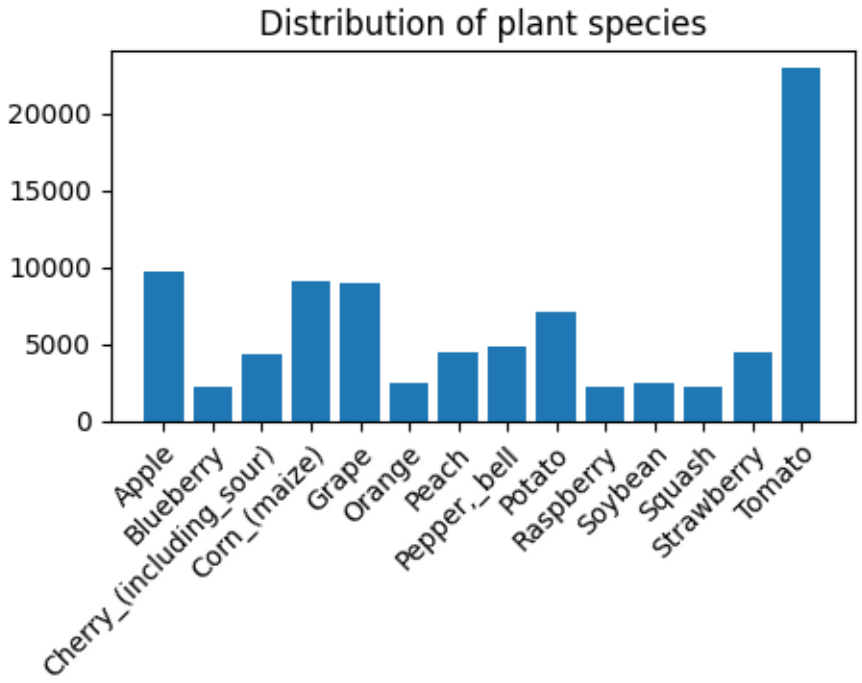
Osserviamo che:

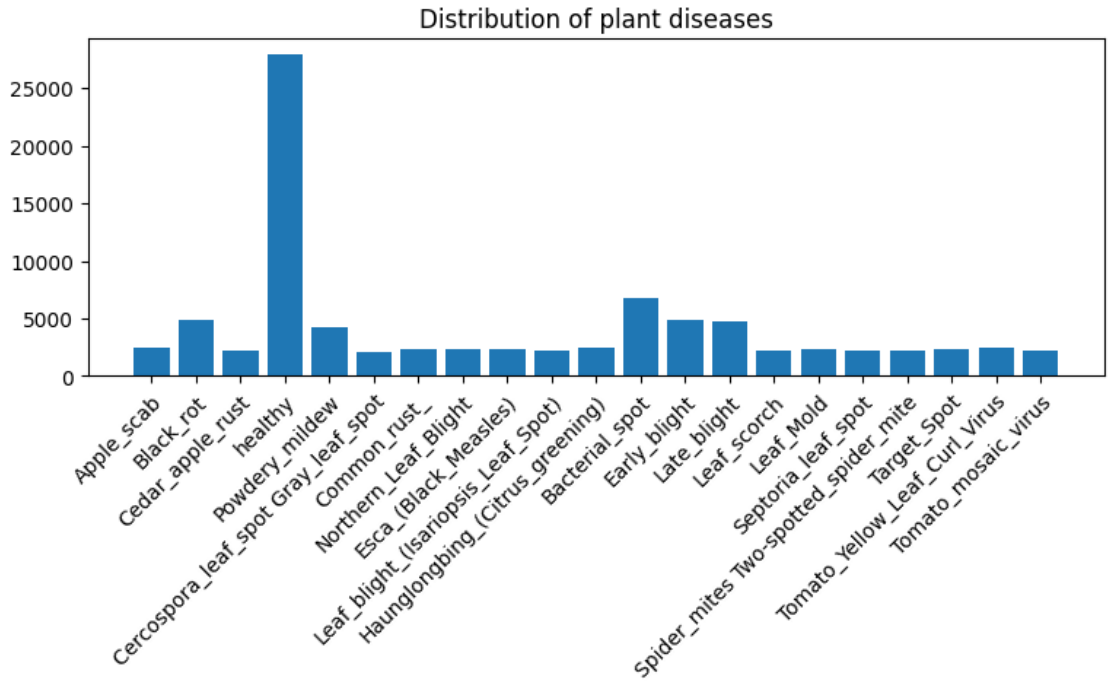
- Il numero di malattie possibili (compresa `healthy`) di ogni pianta è nella maggior parte dei casi nell'intervallo $[2, 4]$.
- Il numero di piante con una data malattia varia tra 1 e 3 nella maggior parte dei casi.
- Le varie classi hanno una distribuzione pressoché uniforme, con circa 2000 elementi per ogni classe.

Visualizziamo ora separatamente gli attributi `specie` e `disease`:

[18]:
```
specie_population = {}
for (specie, disease) in parsed_classes:
    if specie not in specie_population:
        specie_population[specie] = 0
    specie_population[specie] += population[((specie, disease))]

disease_population = {}
for (specie, disease) in parsed_classes:
    if disease not in disease_population:
        disease_population[disease] = 0
    disease_population[disease] += population[((specie, disease))]
```

```
plt.figure(figsize=(5,2.5))
plt.title('Distribution of plant species')
plt.bar(specie_population.keys(), specie_population.values())
plt.xticks(rotation=45, ha='right', rotation_mode='anchor')
plt.show()

plt.figure(figsize=(9,3))
plt.title('Distribution of plant diseases')
plt.bar(disease_population.keys(), disease_population.values())
plt.xticks(rotation=45, ha='right', rotation_mode='anchor')
plt.show()
```





Considerando separatamente le specie e le malattie, osserviamo che la distribuzione non è altrettanto uniforme, con un la specie `Tomato` e la (non) malattia `healthy` che predominano sulle altre.

3 Classificazione

Nel costruire la rete neurale per analizzare il dataset, abbiamo valutato due possibilità. La prima consiste nel costruire due reti differenti e allenarle a riconoscere separatamente la specie della pianta e il tipo della malattia, combinando alla fine il risultato. La seconda possibilità consiste invece nel classificare direttamente i samples nelle 38 combinazioni di specie e malattia. Grazie all'analisi del dataset abbiamo convenuto che il secondo approccio è più conveniente per i seguenti motivi:

- Fattorizzare i label ha poco senso in questo caso dato che la matrice che conta le occorrenze di ogni coppia (`specie`, `malattia`) è molto sparsa: date una specie `s` e una malattia `m`, non è detto (anzi, in genere è poco probabile) che la coppia (`s`, `m`) sia effettivamente una delle 38 etichette del dataset. Allenare due reti diverse a riconoscere separatamente specie e malattia porterebbe al problema supplementare di output non presenti tra le classi originali.

- L'articolo in [1] utilizza un approccio intermedio: crea un'unica rete neurale convoluzionale i cui primi strati si allenano a riconoscere separatamente specie e malattia, ma l'informazione viene combinata negli strati successivi. Non abbiamo scelto di implementare quest'approccio perché sostanzialmente più complicato e, soprattutto, perché i risultati da noi ottenuti sono perfettamente paragonabili in termini di accuratezza.

3.1 Training functions

Come visto a lezione, prepariamo alcune funzioni per l'allenamento della rete neurale. In particolare, le funzioni per calcolare il tempo e l'accuratezza, nonché le vere e proprie procedure di allenamento e validazione:

[19]:
```
def accuracy_fn(y_true, y_pred):
    return (100 * torch.eq(y_true, y_pred).sum().item() / len(y_pred))
```

[20]:
```
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               accuracy_fn):
    train_loss, train_acc = 0, 0

    model.train()
    for batch, (X, y) in enumerate(data_loader):

        X = X.to(device)
        y = y.to(device)

        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        train_loss += loss_fn(y_pred, y)
        train_acc += accuracy_fn(y, y_pred.argmax(dim=1))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    train_loss /= len(data_loader)
    train_acc /= len(data_loader)

    return (train_loss.item(), train_acc)

def valid_step(data_loader: torch.utils.data.DataLoader,
               model: torch.nn.Module,
               loss_fn: torch.nn.Module,
               accuracy_fn):
    valid_loss, valid_acc = 0, 0

    model.eval()
    with torch.inference_mode():
        for batch, (X, y) in enumerate(data_loader):

            X = X.to(device)
            y = y.to(device)
```

```
            y_pred = model(X)
            valid_loss += loss_fn(y_pred, y)
            valid_acc += accuracy_fn(y, y_pred.argmax(dim=1))

        valid_loss /= len(data_loader)
        valid_acc /= len(data_loader)

    return (valid_loss.item(), valid_acc)
```

3.2 Implementazione del modello: rete neurale convoluzionale

Procediamo a costruire una rete convoluzionale, traendo spunto dalla Figura 1 (sinistra) di [1]. In particolare la rete si compone dei seguenti strati (l'input ha dimensione `BATCH_SIZE`$\times 3 \times 64 \times 64$):

- Due convoluzioni con kernel 5×5, padding e stride 1, raddoppiando ogni volta il numero di canali (ora il tensore ha dimensione `BATCH_SIZE`$\times 32 \times 60 \times 60$).
- Un max pooling (ora il tensore ha dimensione `BATCH_SIZE`$\times 32 \times 30 \times 30$).
- Altre convoluzioni con kernel 3×3, padding e stride 1 (ora il tensore ha dimensione `BATCH_SIZE`$\times 256 \times 30 \times 30$).
- Un max pooling (ora il tensore ha dimensione `BATCH_SIZE`$\times 256 \times 15 \times 15$).
- Un average pooling sulla ultime due dimensioni e un flattening (ora il tensore ha dimensione `BATCH_SIZE`$\times 256$).
- Un linear classifier (l'output ha dimensione `BATCH_SIZE`$\times 38$).

[21]:
```
from torch import nn
class AggregateConvolution(nn.Module):
    def __init__(self, input_shape: int, output_shape: int):
        super().__init__()
        self.classifier = nn.Sequential(
            # First convolution
            nn.Conv2d(input_shape, 16, kernel_size = 5, padding = 1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            # Second convolution
            nn.Conv2d(16, 32, kernel_size = 5, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            # Max pooling
            nn.MaxPool2d(kernel_size = 2),
            # Third convolution
            nn.Conv2d(32, 64, kernel_size = 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # Fourth convolution
            nn.Conv2d(64, 256, kernel_size = 3, padding=1),
```

```
            nn.BatchNorm2d(256),
            nn.ReLU(),
            # Final pooling (max+average)
            nn.MaxPool2d(kernel_size = 2),
            nn.AdaptiveAvgPool2d((1,1)),
            # Linear classifier
            nn.Flatten(),
            nn.Linear(in_features = 256, out_features = output_shape),
        )

    def forward(self, x: torch.Tensor):
        x = self.classifier(x)
        return x

torch.manual_seed(0)
aggregate_model_64 = AggregateConvolution(input_shape=3,␣
 ↪output_shape=len(classes))
aggregate_model_64.to(device)
pass
```

Si tratta di un modello con circa 190 000 parametri:

[22]:
```
sum(p.numel() for p in aggregate_model_64.parameters() if p.requires_grad)
```

[22]:
```
190758
```

3.3 Training

Procediamo all'allenamento della rete, valutando a ogni epoca il modello sul `valid_set`. Utilizziamo un *learning rate* esponenziale:

$\texttt{lr} = 0.01 \times 0.95^k$

dove k è il numero dell'epoca. Salviamo inoltre su un file i modelli ottenuti ad ogni epoca, così come l'accuratezza e il valore della *loss function*. Ciò ci consentirà di valutare gli iperparametri migliori, da mettere alla prova in seguito sul test set.

[23]:
```
torch.manual_seed(0)

# Measure time
from timeit import default_timer as timer
start = timer()

model = aggregate_model_64

train_acc = []
valid_acc = []
train_loss = []
```

```
valid_loss = []

# Train and validate model
epochs = 64
for epoch in range(epochs):

    (current_train_loss, current_train_acc) = train_step(
        data_loader = train_dataloader,
        model = model,


        loss_fn = nn.CrossEntropyLoss(),
        optimizer = torch.optim.Adam(params=model.parameters(), lr=0.01*0.
 ↪95**epoch),
        accuracy_fn = accuracy_fn,
    )

    (current_valid_loss, current_valid_acc) = valid_step(data_loader =␣
 ↪valid_dataloader,
        model = model,
        loss_fn = nn.CrossEntropyLoss(),
        accuracy_fn = accuracy_fn,
    )

    train_loss.append(current_train_loss)
    train_acc.append(current_train_acc)
    valid_loss.append(current_valid_loss)
    valid_acc.append(current_valid_acc)

    print(('Epoch ' + str(epoch).rjust(2) + ':  ' +
           f'T. loss = {current_train_loss:.3f},  ' +
           f'T. acc = {current_train_acc:.2f}%,  ' +
           f'V. loss = {current_valid_loss:.3f},  ' +
           f'V. acc = {current_valid_acc:.2f}%'))

    torch.save(model, './aggregate_64_decay_lr_0.95_epoch_'
               + str(epoch) + '.pt')

stop = timer()

torch.save(train_acc, './train_acc.pt')
torch.save(train_loss, './train_loss.pt')
torch.save(valid_acc, './valid_acc.pt')
torch.save(valid_loss, './valid_loss.pt')
torch.save(stop - start, './time_run.pt')
```

```
Epoch  0:  T. loss = 1.337,  T. acc = 60.10%,  V. loss = 4.910,  V. acc = 30.23%
```

```
Epoch  1:  T. loss = 0.554,  T. acc = 82.74%,  V. loss = 0.677,  V. acc = 78.61%
Epoch  2:  T. loss = 0.331,  T. acc = 89.37%,  V. loss = 0.292,  V. acc = 90.34%
Epoch  3:  T. loss = 0.231,  T. acc = 92.63%,  V. loss = 0.785,  V. acc = 78.77%
Epoch  4:  T. loss = 0.177,  T. acc = 94.22%,  V. loss = 0.209,  V. acc = 93.06%
Epoch  5:  T. loss = 0.143,  T. acc = 95.39%,  V. loss = 0.096,  V. acc = 96.90%
Epoch  6:  T. loss = 0.117,  T. acc = 96.18%,  V. loss = 0.149,  V. acc = 95.05%
Epoch  7:  T. loss = 0.098,  T. acc = 96.85%,  V. loss = 0.246,  V. acc = 91.69%
Epoch  8:  T. loss = 0.084,  T. acc = 97.25%,  V. loss = 0.108,  V. acc = 96.44%
Epoch  9:  T. loss = 0.073,  T. acc = 97.49%,  V. loss = 0.173,  V. acc = 94.47%
Epoch 10:  T. loss = 0.065,  T. acc = 97.93%,  V. loss = 0.098,  V. acc = 96.65%
Epoch 11:  T. loss = 0.059,  T. acc = 98.08%,  V. loss = 0.391,  V. acc = 89.81%
Epoch 12:  T. loss = 0.053,  T. acc = 98.27%,  V. loss = 0.077,  V. acc = 97.24%
Epoch 13:  T. loss = 0.044,  T. acc = 98.56%,  V. loss = 0.117,  V. acc = 96.20%
Epoch 14:  T. loss = 0.043,  T. acc = 98.63%,  V. loss = 0.040,  V. acc = 98.75%
Epoch 15:  T. loss = 0.037,  T. acc = 98.79%,  V. loss = 0.046,  V. acc = 98.47%
Epoch 16:  T. loss = 0.032,  T. acc = 98.95%,  V. loss = 0.039,  V. acc = 98.64%
Epoch 17:  T. loss = 0.029,  T. acc = 99.04%,  V. loss = 0.050,  V. acc = 98.34%
Epoch 18:  T. loss = 0.025,  T. acc = 99.13%,  V. loss = 0.041,  V. acc = 98.63%
Epoch 19:  T. loss = 0.025,  T. acc = 99.22%,  V. loss = 0.045,  V. acc = 98.59%
Epoch 20:  T. loss = 0.022,  T. acc = 99.24%,  V. loss = 0.051,  V. acc = 98.45%
Epoch 21:  T. loss = 0.020,  T. acc = 99.39%,  V. loss = 0.047,  V. acc = 98.55%
Epoch 22:  T. loss = 0.018,  T. acc = 99.37%,  V. loss = 0.068,  V. acc = 97.90%
Epoch 23:  T. loss = 0.017,  T. acc = 99.45%,  V. loss = 0.026,  V. acc = 99.13%
Epoch 24:  T. loss = 0.015,  T. acc = 99.52%,  V. loss = 0.030,  V. acc = 99.07%
Epoch 25:  T. loss = 0.014,  T. acc = 99.56%,  V. loss = 0.030,  V. acc = 99.05%
Epoch 26:  T. loss = 0.013,  T. acc = 99.56%,  V. loss = 0.023,  V. acc = 99.30%
Epoch 27:  T. loss = 0.012,  T. acc = 99.64%,  V. loss = 0.040,  V. acc = 98.85%
Epoch 28:  T. loss = 0.011,  T. acc = 99.66%,  V. loss = 0.040,  V. acc = 98.70%
Epoch 29:  T. loss = 0.010,  T. acc = 99.69%,  V. loss = 0.025,  V. acc = 99.23%
Epoch 30:  T. loss = 0.009,  T. acc = 99.72%,  V. loss = 0.031,  V. acc = 99.01%
Epoch 31:  T. loss = 0.009,  T. acc = 99.71%,  V. loss = 0.020,  V. acc = 99.39%
Epoch 32:  T. loss = 0.008,  T. acc = 99.77%,  V. loss = 0.025,  V. acc = 99.22%
Epoch 33:  T. loss = 0.007,  T. acc = 99.80%,  V. loss = 0.026,  V. acc = 99.19%
Epoch 34:  T. loss = 0.007,  T. acc = 99.78%,  V. loss = 0.051,  V. acc = 98.38%
Epoch 35:  T. loss = 0.005,  T. acc = 99.83%,  V. loss = 0.031,  V. acc = 99.06%
Epoch 36:  T. loss = 0.005,  T. acc = 99.84%,  V. loss = 0.026,  V. acc = 99.29%
Epoch 37:  T. loss = 0.005,  T. acc = 99.84%,  V. loss = 0.022,  V. acc = 99.35%
Epoch 38:  T. loss = 0.005,  T. acc = 99.85%,  V. loss = 0.023,  V. acc = 99.31%
Epoch 39:  T. loss = 0.004,  T. acc = 99.87%,  V. loss = 0.021,  V. acc = 99.41%
Epoch 40:  T. loss = 0.004,  T. acc = 99.90%,  V. loss = 0.023,  V. acc = 99.34%
Epoch 41:  T. loss = 0.005,  T. acc = 99.87%,  V. loss = 0.022,  V. acc = 99.39%
Epoch 42:  T. loss = 0.004,  T. acc = 99.89%,  V. loss = 0.020,  V. acc = 99.40%
Epoch 43:  T. loss = 0.003,  T. acc = 99.92%,  V. loss = 0.020,  V. acc = 99.45%
Epoch 44:  T. loss = 0.004,  T. acc = 99.89%,  V. loss = 0.018,  V. acc = 99.45%
Epoch 45:  T. loss = 0.002,  T. acc = 99.93%,  V. loss = 0.019,  V. acc = 99.43%
Epoch 46:  T. loss = 0.003,  T. acc = 99.94%,  V. loss = 0.020,  V. acc = 99.41%
Epoch 47:  T. loss = 0.003,  T. acc = 99.93%,  V. loss = 0.023,  V. acc = 99.40%
Epoch 48:  T. loss = 0.002,  T. acc = 99.94%,  V. loss = 0.018,  V. acc = 99.46%
```

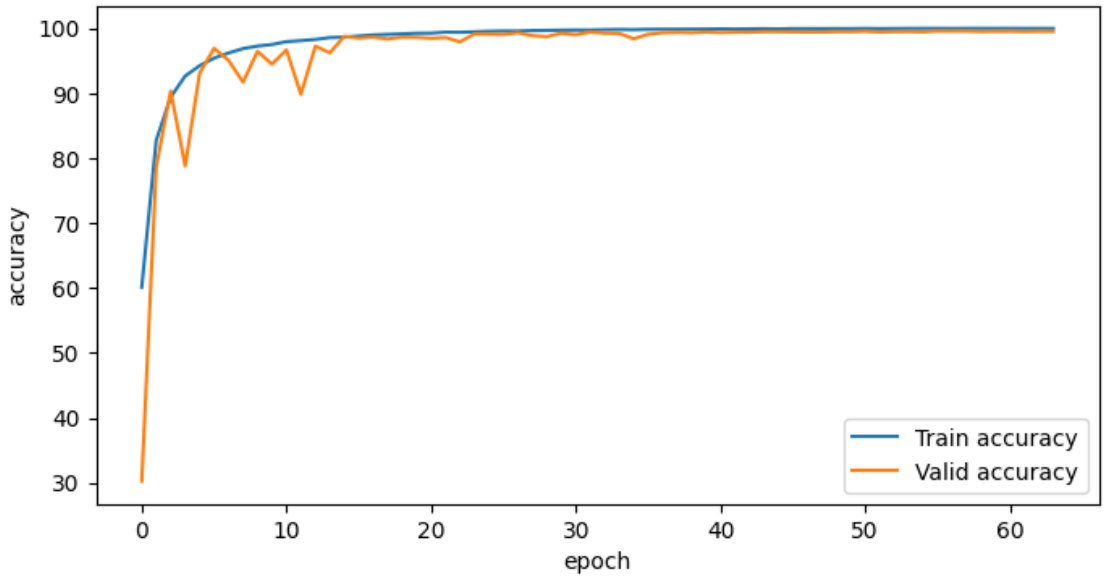
```
Epoch 49:  T. loss = 0.002,  T. acc = 99.94%,  V. loss = 0.020,  V. acc = 99.46%
Epoch 50:  T. loss = 0.002,  T. acc = 99.96%,  V. loss = 0.017,  V. acc = 99.52%
Epoch 51:  T. loss = 0.002,  T. acc = 99.94%,  V. loss = 0.021,  V. acc = 99.41%
Epoch 52:  T. loss = 0.002,  T. acc = 99.95%,  V. loss = 0.019,  V. acc = 99.46%
Epoch 53:  T. loss = 0.002,  T. acc = 99.96%,  V. loss = 0.020,  V. acc = 99.48%
Epoch 54:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.023,  V. acc = 99.43%
Epoch 55:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.018,  V. acc = 99.56%
Epoch 56:  T. loss = 0.002,  T. acc = 99.96%,  V. loss = 0.018,  V. acc = 99.55%
Epoch 57:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.018,  V. acc = 99.55%
Epoch 58:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.018,  V. acc = 99.48%
Epoch 59:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.017,  V. acc = 99.52%
Epoch 60:  T. loss = 0.001,  T. acc = 99.98%,  V. loss = 0.019,  V. acc = 99.51%
Epoch 61:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.019,  V. acc = 99.48%
Epoch 62:  T. loss = 0.001,  T. acc = 99.97%,  V. loss = 0.017,  V. acc = 99.51%
Epoch 63:  T. loss = 0.001,  T. acc = 99.96%,  V. loss = 0.018,  V. acc = 99.50%
```

[36]:
```
dt = int(torch.load("./time_run.pt"))
print(f'Tempo di allenamento (h:m:s): {datetime.timedelta(seconds = dt)}')
```

```
Tempo di allenamento (h:m:s): 1:56:04
```

Osserviamo che, al crescere del numero di epoche, il modello tende ad ottenere accuratezze sempre migliori sia sul *training set*, sia sul *valid set*, senza andare in *overfitting*. Non abbiamo provato ad allenare il modello oltre le 64 epoche, visto il gran tempo computazionale richiesto.

[37]:
```
plt.figure(figsize=(8,4))

train_acc = torch.load('./train_acc.pt')
valid_acc = torch.load('./valid_acc.pt')

plt.plot(train_acc, label='Train accuracy')
plt.plot(valid_acc, label='Valid accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```



4 Evaluation

Scegliamo il modello che ha ottenuto il risultato migliore sul `valid_set` al variare delle epoche, cioè quello corrispondente all'epoca 55:

[38]:
```
best_epoch = valid_acc.index(max(valid_acc))
print(f'Loading model aggregate_64_decay_lr_0.95 of epoch {best_epoch}')
model = torch.load('./aggregate_64_decay_lr_0.95_epoch_' + \
                   str(best_epoch) + '.pt')
model.to(device)
pass
```

```
Loading model aggregate_64_decay_lr_0.95 of epoch 55
```

4.1 Analisi del modello sul `test_set`

4.1.1 Accuratezza

Utilizziamo una funzione simile a `valid_step` in cui però creiamo anche dei vettori numpy contenenti varie forme dell'output del modello sul test set, al fine di applicare gli strumenti di valutazione di `sklearn.metrics`. In particolare:

- `y_true_numpy` è un vettore che contiene le etichette corrette delle immagini del test set (con valori da 0 a 37).
- `y_pred_softmax_numpy` è una matrice il cui elemento (i, j) è la probabilità che l'immagine i sia della classe j predetta dalla rete neruale.
- `y_pred_onehot_numpy` è una matrice il cui la riga i-esima contiene un vettore binario con un 1 nella posizione corrispondente alla classe dell'immagine.

[39]:
```
y_true_numpy = np.zeros(len(test_set), dtype=int)
y_pred_numpy = np.zeros((len(test_set),len(classes)), dtype=float)
y_pred_softmax_numpy = np.zeros((len(test_set),len(classes)))
y_true_onehot_numpy = np.zeros_like(y_pred_softmax_numpy)

test_loss, test_acc = 0, 0

loss_fn = nn.CrossEntropyLoss()
softmax = nn.Softmax(dim=1)

model.eval()
with torch.inference_mode():
    for batch, (X, y) in enumerate(test_dataloader):
        pass

        X = X.to(device)
        y = y.to(device)

        y_pred = model(X)
        test_loss += loss_fn(y_pred, y)
        test_acc += accuracy_fn(y, y_pred.argmax(dim=1))

        y_true_numpy[batch*BATCH_SIZE:(batch+1)*BATCH_SIZE] = y.to('cpu').
 ↪numpy()
        y_pred_softmax_numpy[batch*BATCH_SIZE:(batch+1)*BATCH_SIZE] =␣
 ↪softmax(model(X)).to('cpu').numpy()

    test_loss /= len(test_dataloader)
    test_acc /= len(test_dataloader)

y_pred_numpy = np.argmax(y_pred_softmax_numpy, axis=1)

label_binarizer = LabelBinarizer().fit(y_true_numpy)
y_true_onehot_numpy = label_binarizer.transform(y_true_numpy)

print(f'Test loss: {test_loss}, Test acc: {test_acc}%')
```

```
Test loss: 0.02460133470594883, Test acc: 99.4715909090909%
```

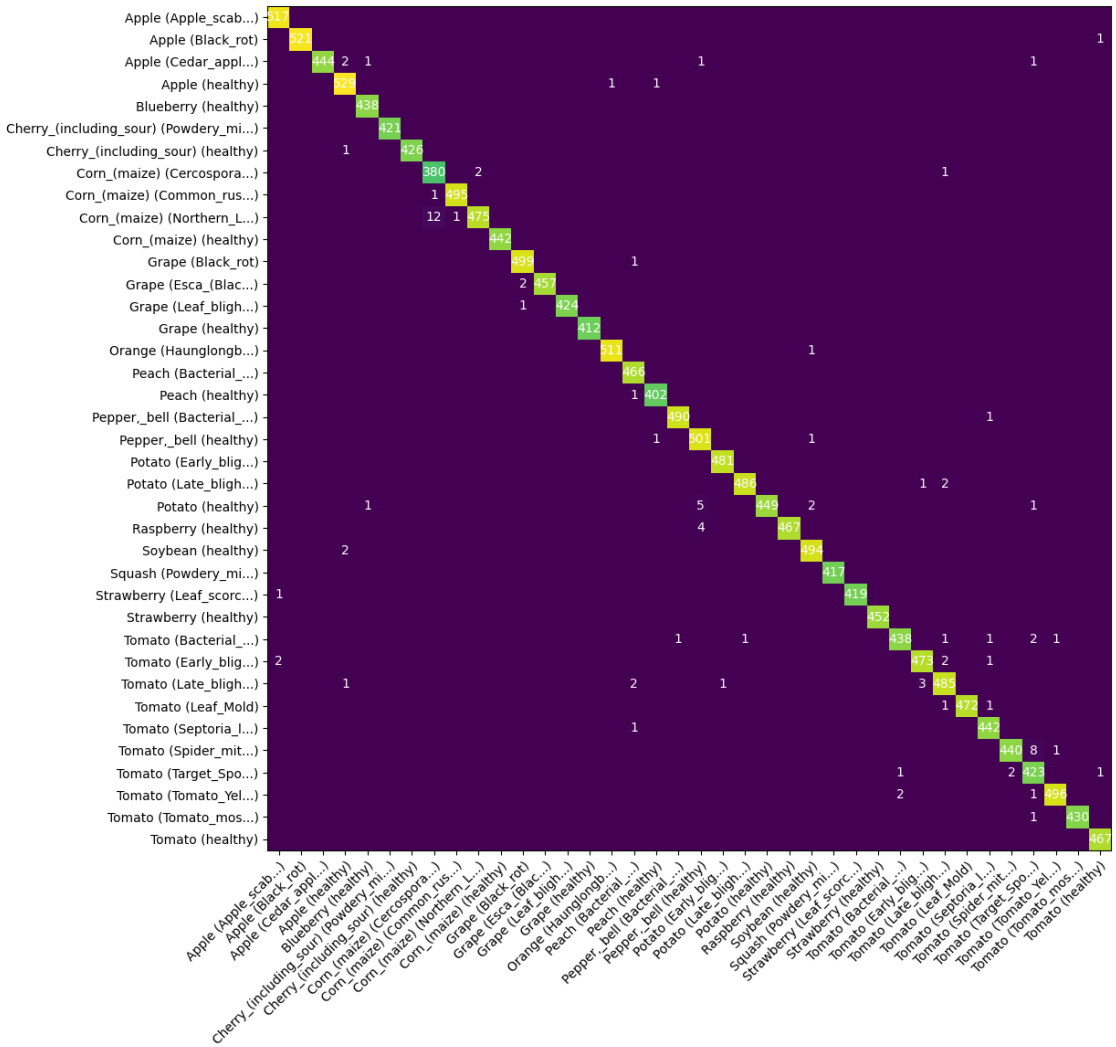
L'accuratezza calcolata è del 99.47%, perfettamente paragonabile con i migliori risultati ottenuti in [1] (v. Tabella 3).

4.1.2 Confusion matrix

Visualizziamo inoltre l'accuratezza del modello tramite una *confusion matrix*:

[40]:
```
def print_parsed_label(l, max_length):
    (specie, disease) = l
```

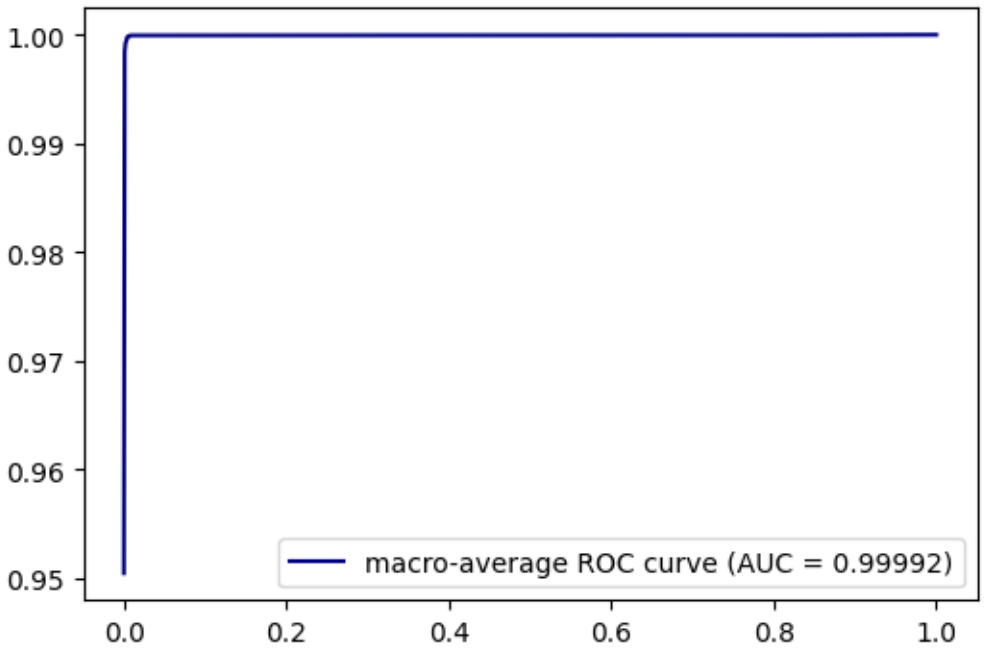
```
    return specie + ' (' + disease[:max_length] + (')' if␣
 ↪len(disease)<max_length else '...)')

plt.figure(figsize = (12,12))
confusion = confusion_matrix(y_true_numpy, y_pred_numpy)
short_classes = [print_parsed_label(parse_class(l), 10) for l in classes]
heatmap(short_classes, short_classes, confusion, xrot=45, vmin=np.
 ↪min(confusion), vmax=np.max(confusion))
```



4.1.3 ROC curve e F1 score

Riportiamo inoltre la ROC curve e l'F1 score del modello. Osserviamo che i risultati sono molto buoni, se comparati con quelli di [1].

[41]:
```
from sklearn.metrics import auc, roc_curve

# store the fpr, tpr, and roc_auc for all averaging strategies
fpr, tpr, roc_auc = dict(), dict(), dict()

for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(y_true_onehot_numpy[:, i],␣
 ↪y_pred_softmax_numpy[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

fpr_grid = np.linspace(0.0, 1.0, 1000)
mean_tpr = np.zeros_like(fpr_grid)
for i in range(len(classes)):
    mean_tpr += np.interp(fpr_grid, fpr[i], tpr[i])  # linear interpolation
mean_tpr /= len(classes)

rocauc_macro = auc(fpr_grid, mean_tpr)

plt.figure(figsize=(6,4))
plt.plot(fpr_grid, mean_tpr, color = "navy",linestyle = "-",
    label = f"macro-average ROC curve (AUC = {rocauc_macro:.5f})",
)
plt.legend()
plt.show()
```



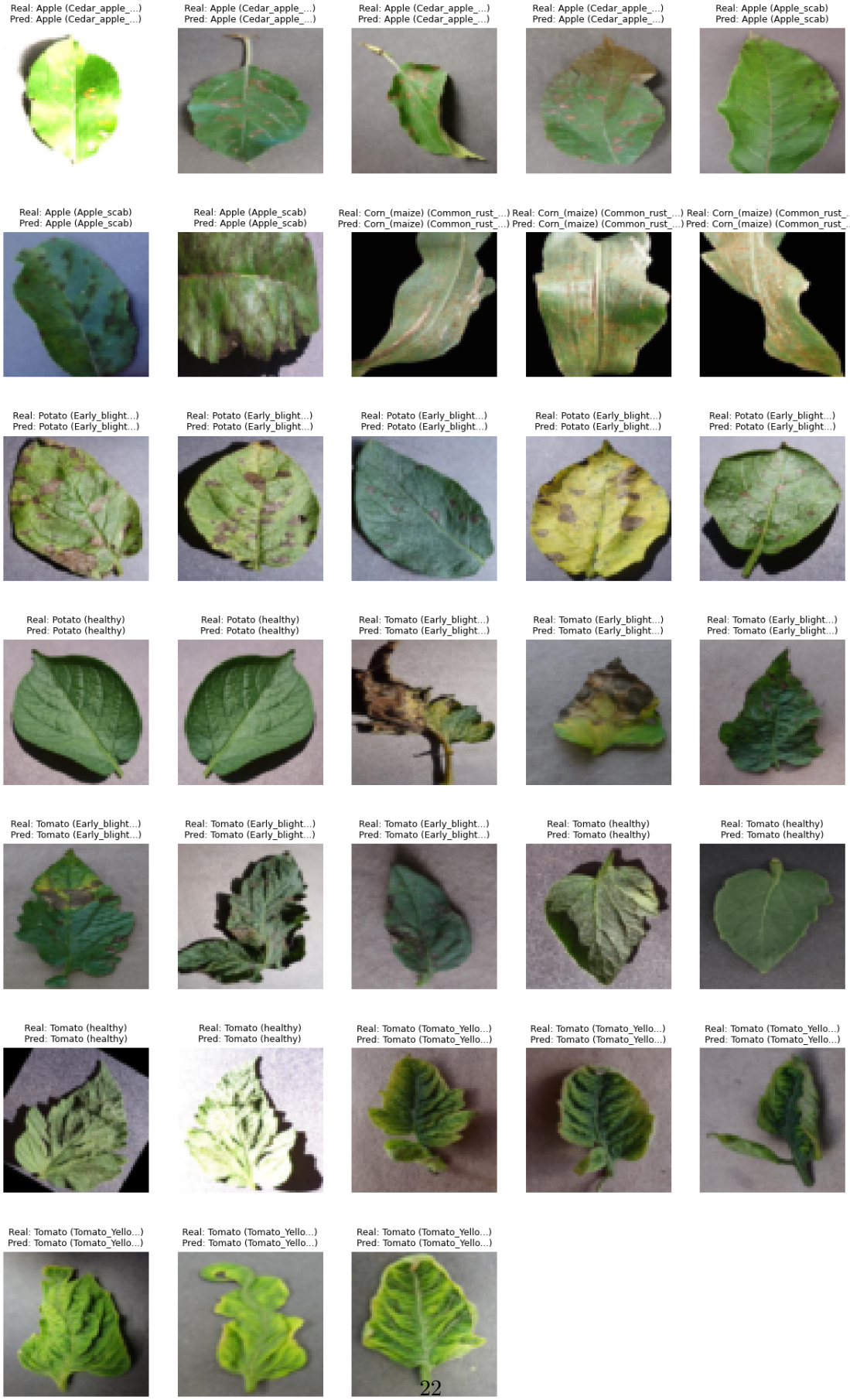
[42]:
```
f1_score(y_true_numpy, y_pred_numpy, average='macro')
```

[42]:
```
0.9946964011740473
```

4.2 Analisi del modello sul `minitest_set`

Per concludere osserviamo che tutte le immagini del `minitest_set` vengono classificate correttamente. Visualizziamo inoltre le immagini in questione:

[43]:
```
fig = plt.figure(figsize=(15,25))

for i, image in enumerate(minitest_set):
    ax = fig.add_subplot(7, 5, i+1)
    print_image(minitest_set[i][0], ax)
    pred = torch.argmax(model(image[0].unsqueeze(0).to(device))).item()
    ax.set_title('Real: ' + print_parsed_label(minitest_labels[i], 12) +␣
 ↪'\nPred: '+ print_parsed_label(parse_class(classes[pred]), 12), fontsize=9)
```


Real: Apple (Cedar_apple_...)
Pred: Apple (Cedar_apple_...)
Real: Apple (Cedar_apple_...)
Pred: Apple (Cedar_apple_...)
Real: Apple (Cedar_apple_...)
Pred: Apple (Cedar_apple_...)
Real: Apple (Cedar_apple_...)
Pred: Apple (Cedar_apple_...)
Real: Apple (Apple_scab)
Pred: Apple (Apple_scab)
Real: Apple (Apple_scab)
Pred: Apple (Apple_scab)
Real: Apple (Apple_scab)
Pred: Apple (Apple_scab)
Real: Corn_(maize) (Common_rust_...)
Pred: Corn_(maize) (Common_rust_...)
Real: Corn_(maize) (Common_rust_...)
Pred: Corn_(maize) (Common_rust_...)
Real: Corn_(maize) (Common_rust_...)
Pred: Corn_(maize) (Common_rust_...)
Real: Potato (Early_blight...)
Pred: Potato (Early_blight...)
Real: Potato (Early_blight...)
Pred: Potato (Early_blight...)
Real: Potato (Early_blight...)
Pred: Potato (Early_blight...)
Real: Potato (Early_blight...)
Pred: Potato (Early_blight...)
Real: Potato (Early_blight...)
Pred: Potato (Early_blight...)
Real: Potato (healthy)
Pred: Potato (healthy)
Real: Potato (healthy)
Pred: Potato (healthy)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (Early_blight...)
Pred: Tomato (Early_blight...)
Real: Tomato (healthy)
Pred: Tomato (healthy)
Real: Tomato (healthy)
Pred: Tomato (healthy)
Real: Tomato (healthy)
Pred: Tomato (healthy)
Real: Tomato (healthy)
Pred: Tomato (healthy)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)
Real: Tomato (Tomato_Yello...)
Pred: Tomato (Tomato_Yello...)

5 Conclusioni

La rete convoluzionale costruita risulta, dai test fatti, ottimamente in grado di svolgere il compito di riconoscere congiuntamente specie e malattia della pianta, confermando le conclusioni tratte dall'analisi dati iniziale. Nei vari esperimenti fatti, gli aspetti che più hanno influito sulla buona accuratezza delle rete sono i seguenti, in parte tratti da [1]:

- Usare convoluzioni con kernel più grande all'inizio (5×5) e più piccolo in seguito (3×3) e, soprattutto, *normalizzare l'output dopo ogni convoluzione*.

- Aumentare progressivamente il numero di canali (orientativamente, raddoppiarli a ogni convoluzione) e ridurre progressivamente la dimensione dell'immagine (*max pooling*).

- Aggiungere un livello di *global average pooling* alla fine che collassa le due *dimensioni spaziali* dell'immagine, lasciando inalterata la dimensione associata ai *canali*. Quest'idea è recentemente molto usata in letteratura, v. [1] o, per esempio [3] e migliora molto il problema dell'overfitting, rispetto a reti convoluzionali seguite da più livelli *fully connected*.

- Utilizzare un *learning rate* che diminuisce nel tempo, che consente una rapida convergenza a un risultato approssimato (già in 5 epoche si supera l'accuratezza del 90%) e una grande precisione nelle epoche successive.

La rete ottenuta in questo modo ha un'accuratezza paragonabile ai migliori risultati di [1], in cui vengono applicate tecniche più sofisticate, come precedentemente riportato. Tuttavia è doveroso specificare che il dataset su cui abbiamo lavorato è un *augmented dataset* rispetto al dataset considerato in [1]: sono presenti molte più immagini rispetto al dataset originale, ottenute tramite trasformazioni come rotazioni, ecc. Naturalmente questo ha influito positivamente sulla qualità della rete qui elaborata.

6 Bibliografia

[1] Schwarz Schuler, J.P., Romani, S., Abdel-Nasser, M., Rashwan, H. and Puig, D. 2022. "Color-Aware Two-Branch DCNN for Efficient Plant Disease Classification." *MENDEL. 28*, 1 (Jun. 2022), 55-62. DOI: https://doi.org/10.13164/mendel.2022.1.055.

[2] Mohanty Sharada P., Hughes David P., Salathé Marcel, "Using Deep Learning for Image-Based Plant Disease Detection." *Frontiers in Plant Science*, 7 (2016), DOI: https://doi.org/10.3389/fpls.2016.01419

[3] A. Kausar, M. Sharif, J. Park and D. R. Shin, "Pure-CNN: A Framework for Fruit Images Classification," *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 2018, pp. 404-408, doi: https://doi.org/10.1109/CSCI46756.2018.00082.