Audio Classification using CNN

In this notebook, we will exploit the features obtainable with torch audio to build a simple audio classifier.

Let's start by importing required libraries.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchaudio
import sys

import matplotlib.pyplot as plt
import IPython.display as ipd

from tqdm import tqdm
```

It's better to use GPU computing to speed-up training.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

Dataset

For this experiment, we are going to use the "SPEECHCOMMANDS" dataset, available on the basic torchaudio datasets (other datasets available here: https://pytorch.org/audio/stable/datasets.html).

Let's create a dedicated class to represents a portion of the SC dataset (either training, validation, or testing).

```
from torchaudio.datasets import SPEECHCOMMANDS
import os


class SubsetSC(SPEECHCOMMANDS):
    def __init__(self, subset: str = None):
        super().__init__("./", download=True)

        def load_list(filename):
            filepath = os.path.join(self._path, filename)
            with open(filepath) as fileobj:
                return [os.path.normpath(os.path.join(self._path,
```

```
line.strip())) for line in fileobj]

        if subset == "validation":
            self._walker = load_list("validation_list.txt")
        elif subset == "testing":
            self._walker = load_list("testing_list.txt")
        elif subset == "training":
            excludes = load_list("validation_list.txt") +
load_list("testing_list.txt")
            excludes = set(excludes)
            self._walker = [w for w in self._walker if w not in
excludes]


# Create training and testing split of the data. We do not use
validation in this tutorial.
train_set = SubsetSC("training")
test_set = SubsetSC("testing")


100%|██████████| 2.26G/2.26G [01:58<00:00, 20.4MB/s]
```

Add support functions (e.g., function to play audio).

```
from IPython.display import Audio

def plot_waveform(waveform):
  waveform_numpy = waveform.t().numpy()
  plt.plot(waveform_numpy);
  plt.show(block=False)

def play_audio(waveform, sample_rate):
  waveform = waveform.numpy()

  num_channels, num_frames = waveform.shape
  if num_channels == 1:
    display(Audio(waveform[0], rate=sample_rate)) # for stereo audio
(two channels)
  elif num_channels == 2:
    display(Audio((waveform[0], waveform[1]), rate=sample_rate))
  else:
    raise ValueError("Waveform with more than 2 channels are not
supported.")
```

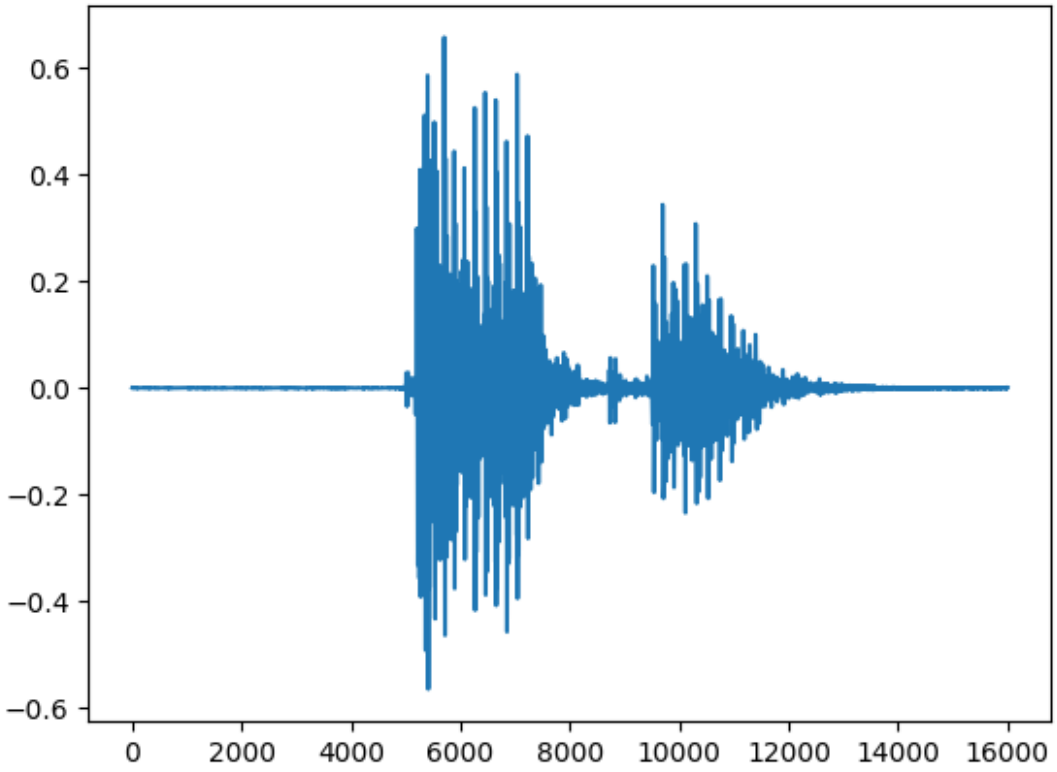
Let's explore a tuple of the dataset. It contains the audio waveform and sample_rate, the target label (supervised), the speaker identifier, and the id of the utterance (command).

```
waveform, sample_rate, label, speaker_id, utterance_number =
train_set[2]
```

```
print(waveform,waveform.size())
print("Sample rate: ",sample_rate)

plot_waveform(waveform)
```

```
tensor([[-6.1035e-05, -1.8311e-04, -2.4414e-04,  ..., -9.1553e-05,
          2.4414e-04,  4.2725e-04]]) torch.Size([1, 16000])
Sample rate:  16000
```




```
labels = sorted(list(set(datapoint[2] for datapoint in train_set)))
labels
```

```
['backward',
 'bed',
 'bird',
 'cat',
 'dog',
 'down',
 'eight',
 'five',
 'follow',
 'forward',
 'four',
 'go',
 'happy',
```

```
 'house',
 'learn',
 'left',
 'marvin',
 'nine',
 'no',
 'off',
 'on',
 'one',
 'right',
 'seven',
 'sheila',
 'six',
 'stop',
 'three',
 'tree',
 'two',
 'up',
 'visual',
 'wow',
 'yes',
 'zero']

print('Total number of entries in training set :', (len(train_set)))

print('Total number of entries in test set :', (len(test_set)))
```

```
Total number of entries in training set : 84843
Total number of entries in test set : 11005
```

We define a resample transformation, we are going to use it later in the training phase. With this we can apply re-sampling using *transform*.

```
import torchaudio.transforms as T
new_sample_rate = 8000
transform = T.Resample(sample_rate, new_sample_rate)

#let's test it on the previously extracted waveform (see previous
cell)
transformed = transform(waveform)
play_audio(transformed,new_sample_rate)
```

```
<IPython.lib.display.Audio object>
```

Let's now build two simple support functions. The former is to obtain the position of a command (word) in the list of labels - note it will return a tensor (see later use). The latter simply returns the command (label) given an input index.

```
def label_to_index(word):
    # Return a tensor with the position of the word into the list of
labels.
    return torch.tensor(labels.index(word))


def index_to_label(index):
    # Return the word in the sorted list of labels
    return labels[index]


word = "visual"
index = label_to_index(word)
word_retrieved = index_to_label(index)

print(word, "-->", index, "-->", word_retrieved)
```

```
visual --> tensor(31) --> visual
```

Building DataLoader

We need to prepare batches of audio tensors (i.e., waveforms). To do so, we create a collate_fn function (see https://pytorch.org/docs/stable/data.html#working-with-collate-fn). Also, audio samples have different lenght (duration) and, therefore, we need to make all of them aligned. So we use the pad_sequence function and add zero padding to the tensors to make them aligned. We need to engineer a little bit the output tensor in light of the output produced by the rnn.pad_sequence function. Input to our function is a list of waveforms.

```
def pad_sequence(batch):
    # Make all tensor in a batch the same length by padding with zeros
    batch = [item.t() for item in batch]
    batch = torch.nn.utils.rnn.pad_sequence(batch, batch_first=True,
padding_value=0.)
    return batch.permute(0, 2, 1)


def collate_fn(batch):

    # A data tuple has the form:
    # waveform, sample_rate, label, speaker_id, utterance_number

    tensors, targets = [], []

    # Gather in lists, and encode labels as indices
    for waveform, _, label, *_ in batch:
        tensors += [waveform]
        targets += [label_to_index(label)]

    # Group the list of tensors into a batched tensor
    tensors = pad_sequence(tensors)
```

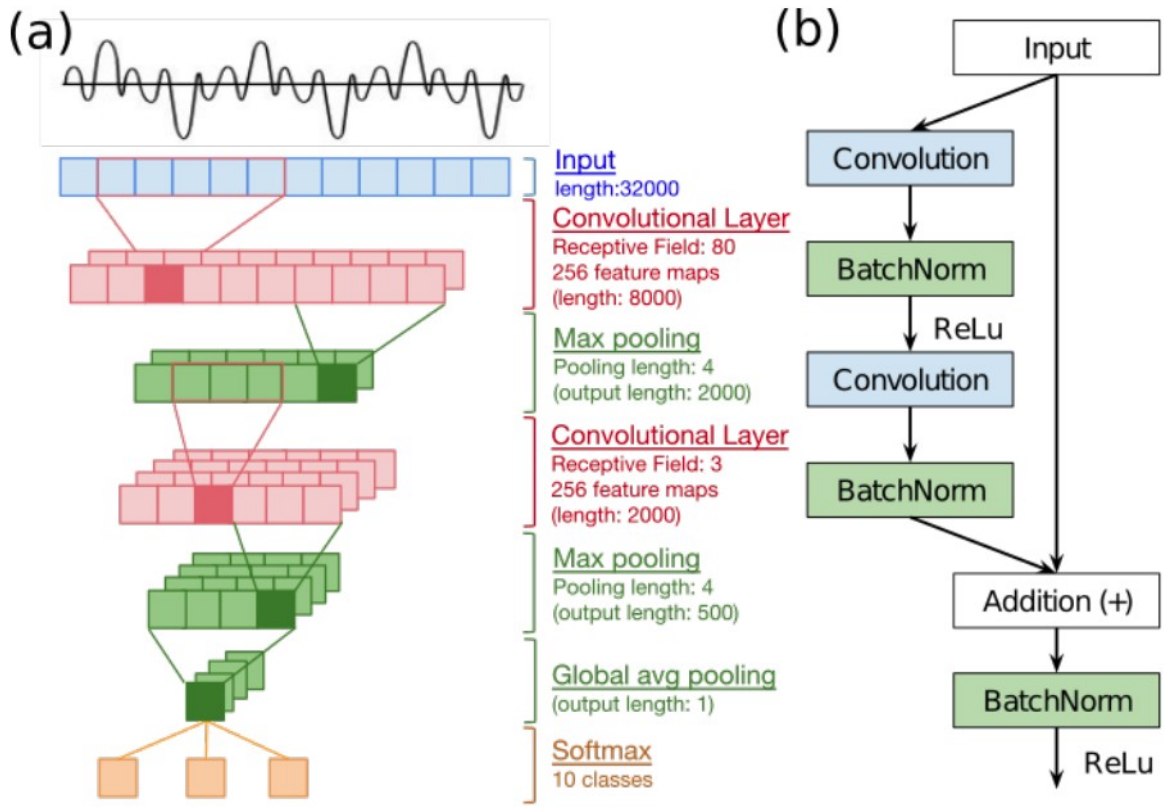
```
    targets = torch.stack(targets)

    return tensors, targets


batch_size = 256 #prima era 256 (accuratezza 74% con 3 epoche), 400
(71% con 3 epoche), 128(78% con 3 epoche)

#Always check which device you are using...
if device == "cuda":
    num_workers = 1
    pin_memory = True
else:
    num_workers = 0
    pin_memory = False

train_loader = torch.utils.data.DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    collate_fn=collate_fn,
    num_workers=num_workers,
    pin_memory=pin_memory,
)
test_loader = torch.utils.data.DataLoader(
    test_set,
    batch_size=batch_size,
    shuffle=False,
    drop_last=False,
    collate_fn=collate_fn,
    num_workers=num_workers,
    pin_memory=pin_memory,
)
```



We are ready to build our model. We are going to use an architecture from the literature. This is known as M5: https://arxiv.org/pdf/1610.00087.pdf

Report: study the different architectures inspect the difference between M3 and M5.

```
class M5(nn.Module):
    def __init__(self, n_input=1, n_output=35, stride=16,
n_channel=32):
        super().__init__()
        self.conv1 = nn.Conv1d(n_input, n_channel, kernel_size=80,
stride=stride)
        self.bn1 = nn.BatchNorm1d(n_channel)
        self.pool1 = nn.MaxPool1d(4)
        self.conv2 = nn.Conv1d(n_channel, n_channel, kernel_size=3)
        self.bn2 = nn.BatchNorm1d(n_channel)
        self.pool2 = nn.MaxPool1d(4)
        self.conv3 = nn.Conv1d(n_channel, 2 * n_channel,
kernel_size=3)
        self.bn3 = nn.BatchNorm1d(2 * n_channel)
        self.pool3 = nn.MaxPool1d(4)
        self.conv4 = nn.Conv1d(2 * n_channel, 2 * n_channel,
kernel_size=3)
        self.bn4 = nn.BatchNorm1d(2 * n_channel)
        self.pool4 = nn.MaxPool1d(4)
        self.fc1 = nn.Linear(2 * n_channel, n_output)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = self.pool2(x)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = self.pool3(x)
        x = self.conv4(x)
        x = F.relu(self.bn4(x))
        x = self.pool4(x)
        x = F.avg_pool1d(x, x.shape[-1])
        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        return F.log_softmax(x, dim=2)


model = M5(n_input=transformed.shape[0], n_output=len(labels))
model.to(device)
print(model)


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)


n = count_parameters(model)
print("Number of parameters: %s" % n)
```

```
M5(
  (conv1): Conv1d(1, 32, kernel_size=(80,), stride=(16,))
  (bn1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool1): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv1d(32, 32, kernel_size=(3,), stride=(1,))
  (bn2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool2): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv1d(32, 64, kernel_size=(3,), stride=(1,))
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool3): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (conv4): Conv1d(64, 64, kernel_size=(3,), stride=(1,))
```

```
  (bn4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool4): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=64, out_features=35, bias=True)
)
Number of parameters: 26915
```

```
class M3(nn.Module):
    def __init__(self, n_input=1, n_output=35, stride=4,
n_channel=128):
        super().__init__()
        self.conv1 = nn.Conv1d(n_input, 2*n_channel, kernel_size=80,
stride=stride)
        self.bn1 = nn.BatchNorm1d(2*n_channel)
        self.pool1 = nn.MaxPool1d(4)
        self.conv2 = nn.Conv1d(2*n_channel, 2*n_channel,
kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm1d(2*n_channel)
        self.pool2 = nn.MaxPool1d(4)
        self.fc1 = nn.Linear(2 * n_channel, n_output)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = self.pool2(x)
        x = F.avg_pool1d(x, x.shape[-1])
        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        return F.log_softmax(x, dim=2)

print('M3 model')
print('Model n_input', transformed.shape[0])
print('Model n_output', len(labels))
print('transformed :', transformed[0])
model_M3 = M3(n_input=transformed.shape[0], n_output=len(labels))
model_M3.to(device)
print(model_M3)

n = count_parameters(model_M3)
print("Number of parameters: %s" % n)
```

```
M3 model
Model n_input 1
Model n_output 35
transformed : tensor([-5.6026e-05, -2.9695e-04, -4.3415e-04,  ..., -
3.9823e-05,
        -6.2212e-05,  2.1935e-04])
M3(
  (conv1): Conv1d(1, 256, kernel_size=(80,), stride=(4,))
  (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool1): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv1d(256, 256, kernel_size=(3,), stride=(1,),
padding=(1,))
  (bn2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool2): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=256, out_features=35, bias=True)
)
Number of parameters: 227619
```

We are goig to use an Adam optimizer and a StepLR reducing the learning reate after every 20 epochs of a factor of 10

```
optimizer = optim.Adam(model.parameters(), lr=0.01,
weight_decay=0.0001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20,
gamma=0.1)  # reduce the learning after 20 epochs by a factor of 10
```

Let's define a "train" function for training our network.

```
def train(model, epoch, log_interval):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):

        data = data.to(device)
        target = target.to(device)

        # apply transform and model on whole batch directly on device
        data = transform(data)
        output = model(data)

        # negative log-likelihood for a tensor of size (batch x 1 x
n_output)
        loss = F.nll_loss(output.squeeze(), target)

        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()

        # print training stats
        if batch_idx % log_interval == 0:
            print(f"Train Epoch: {epoch} [{batch_idx *
len(data)}/{len(train_loader.dataset)} ({100. * batch_idx /
len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

        # update progress bar
        pbar.update(pbar_update)
        # record loss
        losses.append(loss.item())
```

We can build some functions to assess the performance of our network.

```
def number_of_correct(pred, target):
    # count number of correct predictions
    return pred.squeeze().eq(target).sum().item()


def get_likely_index(tensor):
    # find most likely label index for each element in the batch
    return tensor.argmax(dim=-1)


def test(model, epoch):
    model.eval()
    correct = 0
    for data, target in test_loader:

        data = data.to(device)
        target = target.to(device)

        # apply transform and model on whole batch directly on device
        data = transform(data)
        output = model(data)

        pred = get_likely_index(output)
        correct += number_of_correct(pred, target)

        # update progress bar
        pbar.update(pbar_update)

    print(f"\nTest Epoch: {epoch}\tAccuracy:
{correct}/{len(test_loader.dataset)} ({100. * correct /
len(test_loader.dataset):.0f}%)\n")
```

Let's train! The cell below trains our network for n_epoch. Also we use tqdm to build a nice completion bar (https://tqdm.github.io/)

```
#M3 train & validate
log_interval = 20
n_epoch=3

#Use Adam optimiser as used in paper. Weight decay set to 0.0001
#Initially train with learning rate of 0.01. Use a scheduler to
decrease it to 0.001 during training after 20 epochs
optimiser = optim.Adam(model_M3.parameters(), lr=0.01,
weight_decay=0.0001)
scheduler = optim.lr_scheduler.StepLR(optimiser, step_size=20,
gamma=0.1)  # reduce the learning after 20 epochs by a factor of 10

pbar_update = 1 / (len(train_loader) + len(test_loader))
losses = []

# The transform needs to live on the same device as the model and the
data.
transform = transform.to(device)
with tqdm(total=n_epoch) as pbar:
    for epoch in range(1, n_epoch + 1):
        train(model_M3, epoch, log_interval)
        test(model_M3, epoch)
        scheduler.step()

# Let's plot the training loss versus the number of iteration.
# plt.plot(losses);
# plt.title("training loss");
```

```
  0%|          | 0.0026666666666666666/3 [00:02<53:15, 1066.20s/it]

Train Epoch: 1 [0/84843 (0%)]    Loss: 3.615541

  2%|▏         | 0.05599999999999999/3 [00:23<20:31, 418.46s/it]

Train Epoch: 1 [5120/84843 (6%)] Loss: 3.604029

  4%|▍         | 0.10933333333333328/3 [00:44<19:00, 394.67s/it]

Train Epoch: 1 [10240/84843 (12%)]    Loss: 3.631300

  5%|▌         | 0.16266666666666676/3 [01:03<17:02, 360.55s/it]

Train Epoch: 1 [15360/84843 (18%)]    Loss: 3.585282

  7%|▋         | 0.21600000000000033/3 [01:23<17:27, 376.13s/it]

Train Epoch: 1 [20480/84843 (24%)]    Loss: 3.605152

  9%|▊         | 0.2693333333333337/3 [01:44<20:06, 441.87s/it]

Train Epoch: 1 [25600/84843 (30%)]    Loss: 3.615478

 11%|▉         | 0.3226666666666667/3 [02:03<16:14, 364.03s/it]
```

```
Train Epoch: 1 [30720/84843 (36%)]    Loss: 3.611232

 13%|█         | 0.3759999999999997/3 [02:23<18:24, 420.93s/it]

Train Epoch: 1 [35840/84843 (42%)]    Loss: 3.610683

 14%|█         | 0.4293333333333273/3 [02:43<15:36, 364.21s/it]

Train Epoch: 1 [40960/84843 (48%)]    Loss: 3.601424

 16%|█         | 0.4826666666666574/3 [03:03<16:28, 392.76s/it]

Train Epoch: 1 [46080/84843 (54%)]    Loss: 3.602304

 18%|██        | 0.5359999999999995/3 [03:23<14:53, 362.62s/it]

Train Epoch: 1 [51200/84843 (60%)]    Loss: 3.611689

 20%|██        | 0.5893333333333336/3 [03:43<14:57, 372.32s/it]

Train Epoch: 1 [56320/84843 (66%)]    Loss: 3.595884

 21%|██        | 0.6426666666666677/3 [04:03<16:42, 425.35s/it]

Train Epoch: 1 [61440/84843 (72%)]    Loss: 3.610765

 23%|██        | 0.6960000000000018/3 [04:24<14:55, 388.76s/it]

Train Epoch: 1 [66560/84843 (78%)]    Loss: 3.636536

 25%|███       | 0.749333333333336/3 [04:44<15:37, 416.42s/it]

Train Epoch: 1 [71680/84843 (84%)]    Loss: 3.581586

 27%|███       | 0.8026666666666701/3 [05:05<13:48, 376.83s/it]

Train Epoch: 1 [76800/84843 (90%)]    Loss: 3.579956

 29%|███       | 0.8560000000000042/3 [05:25<13:44, 384.71s/it]

Train Epoch: 1 [81920/84843 (96%)]    Loss: 3.618883

 33%|████      | 1.0000000000000062/3 [06:15<11:10,
335.45s/it]/usr/local/lib/python3.10/dist-packages/torch/optim/lr_sche
duler.py:143: UserWarning: Detected call of `lr_scheduler.step()`
before `optimizer.step()`. In PyTorch 1.1.0 and later, you should call
them in the opposite order: `optimizer.step()` before
`lr_scheduler.step()`.  Failure to do this will result in PyTorch
skipping the first value of the learning rate schedule. See more
details at https://pytorch.org/docs/stable/optim.html#how-to-adjust-
learning-rate
  warnings.warn("Detected call of `lr_scheduler.step()` before
`optimizer.step()`. "
```

```
Test Epoch: 1   Accuracy: 196/11005 (2%)

  33%|███       | 1.0026666666666728/3 [06:16<11:30, 345.93s/it]
Train Epoch: 2 [0/84843 (0%)]   Loss: 3.564840
 35%|███       | 1.0560000000000047/3 [06:36<14:06, 435.48s/it]
Train Epoch: 2 [5120/84843 (6%)] Loss: 3.600129
 37%|███       | 1.1093333333333366/3 [06:56<11:39, 369.76s/it]
Train Epoch: 2 [10240/84843 (12%)]    Loss: 3.580287
 39%|███       | 1.1626666666666685/3 [07:16<12:50, 419.24s/it]
Train Epoch: 2 [15360/84843 (18%)]    Loss: 3.581472
 41%|████      | 1.2160000000000004/3 [07:36<10:44, 361.39s/it]
Train Epoch: 2 [20480/84843 (24%)]    Loss: 3.602514
 42%|████      | 1.2693333333333323/3 [07:56<11:45, 407.43s/it]
Train Epoch: 2 [25600/84843 (30%)]    Loss: 3.594836
 44%|████      | 1.3226666666666642/3 [08:17<10:32, 377.14s/it]
Train Epoch: 2 [30720/84843 (36%)]    Loss: 3.600906
 46%|████      | 1.3759999999999961/3 [08:37<10:04, 372.20s/it]
Train Epoch: 2 [35840/84843 (42%)]    Loss: 3.617660
 48%|████      | 1.429333333333328/3 [08:57<11:48, 450.78s/it]
Train Epoch: 2 [40960/84843 (48%)]    Loss: 3.617874
 49%|████      | 1.48266666666666/3 [09:17<09:16, 366.60s/it]
Train Epoch: 2 [46080/84843 (54%)]    Loss: 3.600103
 51%|█████     | 1.5359999999999918/3 [09:37<10:18, 422.15s/it]
Train Epoch: 2 [51200/84843 (60%)]    Loss: 3.611929
 53%|█████     | 1.5893333333333237/3 [09:56<08:33, 364.18s/it]
Train Epoch: 2 [56320/84843 (66%)]    Loss: 3.617229
 55%|█████     | 1.6426666666666556/3 [10:17<08:42, 384.75s/it]
Train Epoch: 2 [61440/84843 (72%)]    Loss: 3.624981
```

```
 57%|█████▋    | 1.6959999999999875/3 [10:36<07:51, 361.61s/it]

Train Epoch: 2 [66560/84843 (78%)]    Loss: 3.597152

 58%|█████▊    | 1.7493333333333194/3 [10:57<07:47, 373.53s/it]

Train Epoch: 2 [71680/84843 (84%)]    Loss: 3.628718

 60%|█████▉    | 1.8026666666666513/3 [11:16<08:19, 417.14s/it]

Train Epoch: 2 [76800/84843 (90%)]    Loss: 3.600456

 62%|██████▏   | 1.8559999999999832/3 [11:36<07:03, 370.35s/it]

Train Epoch: 2 [81920/84843 (96%)]    Loss: 3.600364

 65%|██████▍   | 1.9546666666666472/3 [12:12<05:58, 343.33s/it]
```

```
log_interval = 20
n_epoch = 3
#con 11 epoche e batch size=256 accurattezza 80%

pbar_update = 1 / (len(train_loader) + len(test_loader))
losses = []

# The transform needs to live on the same device as the model and the
data.
transform = transform.to(device)
with tqdm(total=n_epoch) as pbar:
    for epoch in range(1, n_epoch + 1):
        train(model_M3, epoch, log_interval)
        test(model_M3, epoch)
        scheduler.step()

# Let's plot the training loss versus the number of iteration.
# plt.plot(losses);
# plt.title("training loss");
```

```
  0%|          | 0.0026666666666666666/3 [00:02<43:38, 873.45s/it]

Train Epoch: 1 [0/84843 (0%)]    Loss: 3.621367

  2%|▏         | 0.05599999999999999/3 [00:23<18:38, 379.83s/it]

Train Epoch: 1 [5120/84843 (6%)] Loss: 3.605422

  4%|▎         | 0.10933333333333328/3 [00:43<21:34, 447.68s/it]

Train Epoch: 1 [10240/84843 (12%)]    Loss: 3.640987

  5%|▍         | 0.16266666666666676/3 [01:03<17:29, 370.02s/it]

Train Epoch: 1 [15360/84843 (18%)]    Loss: 3.598700
```

```
  7%|▊          | 0.21600000000000033/3 [01:24<19:11, 413.56s/it]
Train Epoch: 1 [20480/84843 (24%)]    Loss: 3.642989
  9%|▉          | 0.2693333333333337/3 [01:43<16:41, 366.71s/it]
Train Epoch: 1 [25600/84843 (30%)]    Loss: 3.604318
 11%|█          | 0.3226666666666667/3 [02:04<17:07, 383.75s/it]
Train Epoch: 1 [30720/84843 (36%)]    Loss: 3.598319
 13%|█▏         | 0.3759999999999997/3 [02:24<19:07, 437.16s/it]
Train Epoch: 1 [35840/84843 (42%)]    Loss: 3.603432
 14%|█▍         | 0.42933333333333273/3 [02:44<15:56, 372.20s/it]
Train Epoch: 1 [40960/84843 (48%)]    Loss: 3.593555
 16%|█▌         | 0.4826666666666574/3 [03:05<17:47, 424.13s/it]
Train Epoch: 1 [46080/84843 (54%)]    Loss: 3.606512
 18%|█▊         | 0.5359999999999995/3 [03:25<15:25, 375.72s/it]
Train Epoch: 1 [51200/84843 (60%)]    Loss: 3.625781
 20%|█▉         | 0.5893333333333336/3 [03:46<15:54, 396.15s/it]
Train Epoch: 1 [56320/84843 (66%)]    Loss: 3.630934
 21%|██         | 0.6426666666666677/3 [04:07<17:33, 446.86s/it]
Train Epoch: 1 [61440/84843 (72%)]    Loss: 3.594502
 23%|██▏        | 0.6960000000000018/3 [04:27<14:21, 373.82s/it]
Train Epoch: 1 [66560/84843 (78%)]    Loss: 3.607289
 25%|██▍        | 0.749333333333336/3 [04:48<15:45, 420.04s/it]
Train Epoch: 1 [71680/84843 (84%)]    Loss: 3.594518
 27%|██▋        | 0.8026666666666701/3 [05:08<13:33, 370.24s/it]
Train Epoch: 1 [76800/84843 (90%)]    Loss: 3.603799
 29%|██▊        | 0.8560000000000042/3 [05:29<13:45, 385.08s/it]
Train Epoch: 1 [81920/84843 (96%)]    Loss: 3.599305
 33%|███        | 1.0000000000000062/3 [06:19<11:21, 340.85s/it]
```

```
Test Epoch: 1   Accuracy: 241/11005 (2%)

  33%|███       | 1.0026666666666728/3 [06:20<11:40, 350.94s/it]
Train Epoch: 2 [0/84843 (0%)]   Loss: 3.629742
 35%|███       | 1.0560000000000047/3 [06:41<14:57, 461.82s/it]
Train Epoch: 2 [5120/84843 (6%)] Loss: 3.603289
 37%|███       | 1.1093333333333366/3 [07:01<11:44, 372.40s/it]
Train Epoch: 2 [10240/84843 (12%)]    Loss: 3.607699
 39%|████      | 1.1626666666666685/3 [07:21<12:29, 408.18s/it]
Train Epoch: 2 [15360/84843 (18%)]    Loss: 3.596784
 41%|████      | 1.2160000000000004/3 [07:41<11:09, 375.05s/it]
Train Epoch: 2 [20480/84843 (24%)]    Loss: 3.605073
 42%|████      | 1.2693333333333323/3 [08:02<11:03, 383.23s/it]
Train Epoch: 2 [25600/84843 (30%)]    Loss: 3.601493
 44%|████      | 1.3226666666666642/3 [08:22<11:50, 423.47s/it]
Train Epoch: 2 [30720/84843 (36%)]    Loss: 3.611003
 46%|████      | 1.3759999999999961/3 [08:42<10:15, 378.99s/it]
Train Epoch: 2 [35840/84843 (42%)]    Loss: 3.617035
 48%|████      | 1.429333333333328/3 [09:03<10:59, 419.92s/it]
Train Epoch: 2 [40960/84843 (48%)]    Loss: 3.618627
 49%|████      | 1.48266666666666/3 [09:23<09:16, 367.01s/it]
Train Epoch: 2 [46080/84843 (54%)]    Loss: 3.575739
 51%|████      | 1.5359999999999918/3 [09:44<09:27, 387.90s/it]
Train Epoch: 2 [51200/84843 (60%)]    Loss: 3.622574
 53%|████      | 1.5893333333333237/3 [10:04<09:41, 412.54s/it]
Train Epoch: 2 [56320/84843 (66%)]    Loss: 3.591389
 55%|████      | 1.6426666666666556/3 [10:24<08:30, 376.35s/it]
Train Epoch: 2 [61440/84843 (72%)]    Loss: 3.592136
```

```
 57%|██████    | 1.6959999999999875/3 [10:45<09:19, 428.92s/it]
Train Epoch: 2 [66560/84843 (78%)]    Loss: 3.612226
 58%|██████    | 1.7493333333333194/3 [11:04<07:41, 369.01s/it]
Train Epoch: 2 [71680/84843 (84%)]    Loss: 3.633183
 60%|██████    | 1.8026666666666513/3 [11:25<07:48, 391.63s/it]
Train Epoch: 2 [76800/84843 (90%)]    Loss: 3.623835
 62%|██████    | 1.8559999999999832/3 [11:44<06:56, 363.84s/it]
Train Epoch: 2 [81920/84843 (96%)]    Loss: 3.618730
 67%|███████   | 1.9999999999999793/3 [12:35<05:19, 319.15s/it]

Test Epoch: 2   Accuracy: 237/11005 (2%)

  67%|███████   | 2.002666666666646/3 [12:36<05:38, 339.71s/it]
Train Epoch: 3 [0/84843 (0%)]   Loss: 3.612593
 69%|███████   | 2.0559999999999823/3 [12:56<05:55, 376.90s/it]
Train Epoch: 3 [5120/84843 (6%)] Loss: 3.631214
 70%|███████   | 2.1093333333333186/3 [13:16<06:36, 445.62s/it]
Train Epoch: 3 [10240/84843 (12%)]    Loss: 3.556427
 72%|███████   | 2.162666666666655/3 [13:36<05:04, 363.25s/it]
Train Epoch: 3 [15360/84843 (18%)]    Loss: 3.622924
 74%|███████   | 2.2159999999999913/3 [13:56<05:19, 408.11s/it]
Train Epoch: 3 [20480/84843 (24%)]    Loss: 3.606650
 76%|███████   | 2.2693333333333277/3 [14:15<04:28, 367.58s/it]
Train Epoch: 3 [25600/84843 (30%)]    Loss: 3.606331
 77%|███████   | 2.322666666666664/3 [14:36<04:19, 383.05s/it]
Train Epoch: 3 [30720/84843 (36%)]    Loss: 3.602437
 79%|███████   | 2.3760000000000003/3 [14:55<04:00, 385.11s/it]
Train Epoch: 3 [35840/84843 (42%)]    Loss: 3.591383
 81%|███████   | 2.4293333333333367/3 [15:16<03:33, 373.59s/it]
```

```
Train Epoch: 3 [40960/84843 (48%)]    Loss: 3.618559

 83%|████████  | 2.482666666666673/3 [15:36<03:50, 446.29s/it]

Train Epoch: 3 [46080/84843 (54%)]    Loss: 3.602975

 85%|████████  | 2.5360000000000094/3 [15:55<02:51, 370.68s/it]

Train Epoch: 3 [51200/84843 (60%)]    Loss: 3.615752

 86%|████████  | 2.5893333333333457/3 [16:16<02:48, 409.56s/it]

Train Epoch: 3 [56320/84843 (66%)]    Loss: 3.596627

 88%|████████  | 2.642666666666682/3 [16:35<02:08, 359.76s/it]

Train Epoch: 3 [61440/84843 (72%)]    Loss: 3.600069

 90%|████████  | 2.6960000000000184/3 [16:55<02:00, 397.87s/it]

Train Epoch: 3 [66560/84843 (78%)]    Loss: 3.612092

 92%|████████  | 2.7493333333333547/3 [17:15<01:32, 367.99s/it]

Train Epoch: 3 [71680/84843 (84%)]    Loss: 3.638482

 93%|████████  | 2.802666666666691/3 [17:35<01:13, 371.94s/it]

Train Epoch: 3 [76800/84843 (90%)]    Loss: 3.591575

 95%|████████  | 2.8560000000000274/3 [17:55<01:04, 450.54s/it]

Train Epoch: 3 [81920/84843 (96%)]    Loss: 3.631505

100%|████████| 2.9973333333333687/3 [18:43<00:00,
322.51s/it]/usr/local/lib/python3.10/dist-packages/tqdm/std.py:636:
TqdmWarning: clamping frac to range [0, 1]
  full_bar = Bar(frac,
100%|████████| 3.0000000000000355/3 [18:45<00:00, 375.01s/it]


Test Epoch: 3   Accuracy: 241/11005 (2%)
```

```python
def count_confusion_matrix(model, data_loader):
    model.eval()
    tp, fp, tn, fn = 0, 0, 0, 0
    with torch.no_grad():
        for data, target in data_loader:
            data = data.to(device)
            target = target.to(device)
            # apply transform and model on the whole batch directly on
```

```
the device
            data = transform(data)
            output = model(data)
            pred = get_likely_index(output)

            # Calculate TP, FP, TN, FN
            tp += ((pred == 1) & (target == 1)).sum().item()
            fp += ((pred == 1) & (target == 0)).sum().item()
            tn += ((pred == 0) & (target == 0)).sum().item()
            fn += ((pred == 0) & (target == 1)).sum().item()

    return tp, fp, tn, fn

# Calculate counts for the training set
tp_train, fp_train, tn_train, fn_train =
count_confusion_matrix(model_M3, train_loader)

# Calculate counts for the test set
tp_test, fp_test, tn_test, fn_test = count_confusion_matrix(model_M3,
test_loader)

# Print counts
print("Training Set:")
print("True Positives (TP):", tp_train)
print("False Positives (FP):", fp_train)
print("True Negatives (TN):", tn_train)
print("False Negatives (FN):", fn_train)
print("\nTest Set:")
print("True Positives (TP):", tp_test)
print("False Positives (FP):", fp_test)
print("True Negatives (TN):", tn_test)
print("False Negatives (FN):", fn_test)
```

```
Training Set:
True Positives (TP): 0
False Positives (FP): 0
True Negatives (TN): 0
False Negatives (FN): 0

Test Set:
True Positives (TP): 0
False Positives (FP): 0
True Negatives (TN): 0
False Negatives (FN): 0
```

```
log_interval = 20
n_epoch = 3
#con 11 epoche e batch size=256 accurattezza 80%

pbar_update = 1 / (len(train_loader) + len(test_loader))
```

```
losses = []

# The transform needs to live on the same device as the model and the
data.
transform = transform.to(device)
with tqdm(total=n_epoch) as pbar:
    for epoch in range(1, n_epoch + 1):
        train(model, epoch, log_interval)
        test(model, epoch)
        scheduler.step()

# Let's plot the training loss versus the number of iteration.
# plt.plot(losses);
# plt.title("training loss");
```

```
  0%|          | 0.0026666666666666666/3 [00:01<30:43, 615.04s/it]
Train Epoch: 1 [0/84843 (0%)]    Loss: 3.790477
  2%|          | 0.05599999999999999/3 [00:19<16:21, 333.44s/it]
Train Epoch: 1 [5120/84843 (6%)] Loss: 3.048852
  4%|▏         | 0.10933333333333328/3 [00:38<18:45, 389.49s/it]
Train Epoch: 1 [10240/84843 (12%)]    Loss: 2.570973
  5%|▏         | 0.16266666666666676/3 [00:56<15:51, 335.52s/it]
Train Epoch: 1 [15360/84843 (18%)]    Loss: 2.277411
  7%|▏         | 0.21600000000000033/3 [01:14<14:59, 323.17s/it]
Train Epoch: 1 [20480/84843 (24%)]    Loss: 2.050221
  9%|▏         | 0.2693333333333337/3 [01:33<15:56, 350.43s/it]
Train Epoch: 1 [25600/84843 (30%)]    Loss: 1.863413
 11%|▏         | 0.3226666666666667/3 [01:52<14:30, 325.08s/it]
Train Epoch: 1 [30720/84843 (36%)]    Loss: 1.911834
 13%|▏         | 0.3759999999999997/3 [02:11<15:27, 353.46s/it]
Train Epoch: 1 [35840/84843 (42%)]    Loss: 1.695160
 14%|▏         | 0.42933333333333273/3 [02:29<14:15, 332.72s/it]
Train Epoch: 1 [40960/84843 (48%)]    Loss: 1.840236
 16%|▏         | 0.48266666666666574/3 [02:48<15:30, 369.52s/it]
Train Epoch: 1 [46080/84843 (54%)]    Loss: 1.641245
```

```
 18%|█         | 0.5359999999999995/3 [03:06<13:25, 326.80s/it]
Train Epoch: 1 [51200/84843 (60%)]    Loss: 1.559800
 20%|█         | 0.5893333333333336/3 [03:25<16:59, 422.75s/it]
Train Epoch: 1 [56320/84843 (66%)]    Loss: 1.461332
 21%|██        | 0.6426666666666677/3 [03:43<13:08, 334.64s/it]
Train Epoch: 1 [61440/84843 (72%)]    Loss: 1.356640
 23%|██        | 0.6960000000000018/3 [04:01<14:05, 367.02s/it]
Train Epoch: 1 [66560/84843 (78%)]    Loss: 1.059988
 25%|██        | 0.749333333333336/3 [04:20<12:46, 340.72s/it]
Train Epoch: 1 [71680/84843 (84%)]    Loss: 1.160042
 27%|██        | 0.8026666666666701/3 [04:38<12:01, 328.17s/it]
Train Epoch: 1 [76800/84843 (90%)]    Loss: 1.009504
 29%|██        | 0.8560000000000042/3 [04:57<12:37, 353.09s/it]
Train Epoch: 1 [81920/84843 (96%)]    Loss: 1.138900
 33%|███       | 1.0000000000000062/3 [05:44<13:03, 391.92s/it]

Test Epoch: 1   Accuracy: 7137/11005 (65%)

  33%|███       | 1.0026666666666728/3 [05:45<12:41, 381.26s/it]
Train Epoch: 2 [0/84843 (0%)]    Loss: 1.034717
 35%|███       | 1.0560000000000047/3 [06:03<10:47, 333.19s/it]
Train Epoch: 2 [5120/84843 (6%)] Loss: 1.092970
 37%|███       | 1.1093333333333366/3 [06:22<12:14, 388.70s/it]
Train Epoch: 2 [10240/84843 (12%)]    Loss: 1.062043
 39%|███       | 1.1626666666666685/3 [06:40<10:13, 333.71s/it]
Train Epoch: 2 [15360/84843 (18%)]    Loss: 1.036001
 41%|████      | 1.2160000000000004/3 [06:58<10:00, 336.41s/it]
Train Epoch: 2 [20480/84843 (24%)]    Loss: 0.923860
 42%|████      | 1.2693333333333323/3 [07:17<09:42, 336.69s/it]
```

```
Train Epoch: 2 [25600/84843 (30%)]    Loss: 1.022502
 44%|████      | 1.3226666666666642/3 [07:35<09:05, 325.39s/it]
Train Epoch: 2 [30720/84843 (36%)]    Loss: 1.059344
 46%|████      | 1.3759999999999961/3 [07:54<09:34, 354.01s/it]
Train Epoch: 2 [35840/84843 (42%)]    Loss: 1.076034
 48%|████      | 1.429333333333328/3 [08:12<08:32, 326.27s/it]
Train Epoch: 2 [40960/84843 (48%)]    Loss: 1.231857
 49%|████      | 1.48266666666666/3 [08:31<09:42, 384.10s/it]
Train Epoch: 2 [46080/84843 (54%)]    Loss: 0.969934
 51%|████      | 1.5359999999999918/3 [08:49<08:03, 329.96s/it]
Train Epoch: 2 [51200/84843 (60%)]    Loss: 0.991519
 53%|████      | 1.5893333333333237/3 [09:07<09:45, 415.20s/it]
Train Epoch: 2 [56320/84843 (66%)]    Loss: 0.976233
 55%|████      | 1.6426666666666556/3 [09:25<07:39, 338.86s/it]
Train Epoch: 2 [61440/84843 (72%)]    Loss: 0.748306
 57%|████      | 1.6959999999999875/3 [09:44<07:42, 354.40s/it]
Train Epoch: 2 [66560/84843 (78%)]    Loss: 0.796559
 58%|████      | 1.7493333333333194/3 [10:02<07:12, 346.00s/it]
Train Epoch: 2 [71680/84843 (84%)]    Loss: 0.826512
 60%|████      | 1.8026666666666513/3 [10:20<06:28, 324.31s/it]
Train Epoch: 2 [76800/84843 (90%)]    Loss: 0.911608
 62%|████      | 1.8559999999999832/3 [10:39<06:40, 349.74s/it]
Train Epoch: 2 [81920/84843 (96%)]    Loss: 0.790906
 67%|████      | 1.9999999999999793/3 [11:26<06:16, 376.06s/it]

Test Epoch: 2   Accuracy: 7722/11005 (70%)

  67%|████      | 2.002666666666646/3 [11:27<06:26, 387.56s/it]
Train Epoch: 3 [0/84843 (0%)]    Loss: 0.908686
```

```
 69%|██████▉   | 2.0559999999999823/3 [11:45<05:13, 331.74s/it]
Train Epoch: 3 [5120/84843 (6%)] Loss: 0.817632
 70%|███████   | 2.1093333333333186/3 [12:03<05:38, 380.44s/it]
Train Epoch: 3 [10240/84843 (12%)]    Loss: 0.907780
 72%|███████   | 2.162666666666655/3 [12:22<04:43, 338.60s/it]
Train Epoch: 3 [15360/84843 (18%)]    Loss: 0.862680
 74%|███████▍  | 2.2159999999999913/3 [12:40<04:14, 324.29s/it]
Train Epoch: 3 [20480/84843 (24%)]    Loss: 0.856575
 76%|███████▌  | 2.2693333333333277/3 [12:59<04:15, 349.88s/it]
Train Epoch: 3 [25600/84843 (30%)]    Loss: 0.883612
 77%|███████▋  | 2.322666666666664/3 [13:17<03:40, 324.99s/it]
Train Epoch: 3 [30720/84843 (36%)]    Loss: 0.708188
 79%|███████▉  | 2.3760000000000003/3 [13:36<03:48, 365.71s/it]
Train Epoch: 3 [35840/84843 (42%)]    Loss: 0.759057
 81%|████████  | 2.4293333333333367/3 [13:53<03:06, 325.98s/it]
Train Epoch: 3 [40960/84843 (48%)]    Loss: 0.809799
 83%|████████▎ | 2.482666666666673/3 [14:12<03:26, 398.41s/it]
Train Epoch: 3 [46080/84843 (54%)]    Loss: 0.994512
 85%|████████▍ | 2.5360000000000094/3 [14:30<02:33, 330.75s/it]
Train Epoch: 3 [51200/84843 (60%)]    Loss: 0.572577
 86%|████████▋ | 2.5893333333333457/3 [14:49<02:33, 372.84s/it]
Train Epoch: 3 [56320/84843 (66%)]    Loss: 0.905681
 88%|████████▊ | 2.642666666666682/3 [15:07<02:01, 340.56s/it]
Train Epoch: 3 [61440/84843 (72%)]    Loss: 0.923929
 90%|████████▉ | 2.6960000000000184/3 [15:25<01:39, 325.70s/it]
Train Epoch: 3 [66560/84843 (78%)]    Loss: 0.722016
 92%|█████████▏| 2.7493333333333547/3 [15:44<01:26, 343.49s/it]
Train Epoch: 3 [71680/84843 (84%)]    Loss: 0.797123
```

```
 93%|█████████ | 2.802666666666691/3 [16:02<01:06, 335.05s/it]

Train Epoch: 3 [76800/84843 (90%)]    Loss: 0.744030

 95%|█████████ | 2.8560000000000274/3 [16:21<00:52, 362.85s/it]

Train Epoch: 3 [81920/84843 (96%)]    Loss: 0.868759

100%|██████████| 3.0000000000000355/3 [17:07<00:00, 342.54s/it]


Test Epoch: 3   Accuracy: 8197/11005 (74%)
```

```python
def count_confusion_matrix(model, data_loader):
    model.eval()
    tp, fp, tn, fn = 0, 0, 0, 0
    with torch.no_grad():
        for data, target in data_loader:
            data = data.to(device)
            target = target.to(device)
            # apply transform and model on the whole batch directly on
the device
            data = transform(data)
            output = model(data)
            pred = get_likely_index(output)

            # Calculate TP, FP, TN, FN
            tp += ((pred == 1) & (target == 1)).sum().item()
            fp += ((pred == 1) & (target == 0)).sum().item()
            tn += ((pred == 0) & (target == 0)).sum().item()
            fn += ((pred == 0) & (target == 1)).sum().item()

    return tp, fp, tn, fn

# Calculate counts for the training set
tp_train, fp_train, tn_train, fn_train = count_confusion_matrix(model,
train_loader)

# Calculate counts for the test set
tp_test, fp_test, tn_test, fn_test = count_confusion_matrix(model,
test_loader)

# Print counts
print("Training Set:")
print("True Positives (TP):", tp_train)
print("False Positives (FP):", fp_train)
print("True Negatives (TN):", tn_train)
print("False Negatives (FN):", fn_train)
print("\nTest Set:")
```

```
print("True Positives (TP):", tp_test)
print("False Positives (FP):", fp_test)
print("True Negatives (TN):", tn_test)
print("False Negatives (FN):", fn_test)
```

```
Training Set:
True Positives (TP): 8173
False Positives (FP): 6039
True Negatives (TN): 6496
False Negatives (FN): 6259

Test Set:
True Positives (TP): 21191
False Positives (FP): 1292
True Negatives (TN): 11272
False Negatives (FN): 1404
```

Report: inspect performance on the different classes. Refine the model and the training strategy to increase them.

Finally, we can use our model for making commands' classification. See the following function.

```
def predict(tensor):
    # Use the model to predict the label of the waveform
    tensor = tensor.to(device)
    tensor = transform(tensor)
    tensor = model(tensor.unsqueeze(0))
    tensor = get_likely_index(tensor)
    tensor = index_to_label(tensor.squeeze())
    return tensor


waveform, sample_rate, utterance, *_ = train_set[-1]
play_audio(waveform, sample_rate)

print(f"Expected: {utterance}. Predicted: {predict(waveform)}.")
```

```
<IPython.lib.display.Audio object>

Expected: zero. Predicted: zero.
```

Report Suggestions:

1. Understand M5 architecture, compare with M3 (see reference paper).
2. Improve model performance (74% accuracy...can we boost it? How?)
3. Assess the performance on the single classes (confusion matrix). What are the main findings?
4. How can we refine this model?