

## ▼ Segmentacija Ehokardiograma

### Računska inteligencija - seminarski rad

Mislav Ivanda

Ivan Lukšić

Roko Smoljić

## ▼ Dataset

---

Naziv: EchoNet-Dynamic

Dataset link: <https://echonet.github.io/dynamic/>

Sadržaj: 10,030 ehokardiograma snimljenih između 2016. i 2018. u sveučilišnoj bolnici sveučilišta Stanford. Snimke su rezolucije 112 x 112 piksela.

## ▼ Drive

```
#@title Drive
from google.colab import drive
drive.mount('/content/drive/', force_remount=True)
```

Mounted at /content/drive/

```
!ls '/content/drive/My Drive/EchoNet-Dynamic'
```

```
  FileList.csv  Videos  VolumeTracings.csv
```

```
!cp -r '/content/drive/My Drive/dynamic' dynamic
```

```
!pip install '/content/dynamic'
```

```
%load_ext autoreload
%autoreload 2
```

```
import echonet
echonet
```

```
!nvidia-smi
```

```
Mon Jun 20 10:22:33 2022
```

```
+-----+  
| NVIDIA-SMI 460.32.03     Driver Version: 460.32.03     CUDA Version: 11.2 |  
+-----+  
| GPU  Name      Persistence-M| Bus-Id     Disp.A  | Volatile Uncorr. ECC |  
| Fan  Temp  Perf  Pwr:Usage/Cap|          Memory-Usage | GPU-Util  Compute M. |  
|                               |                         |                MIG M. |  
+=====+=====+=====+=====+=====+=====+=====+=====+  
| 0  Tesla P100-PCIE... Off | 00000000:00:04.0 Off |           0 |  
| N/A   39C    P0    27W / 250W |           0MiB / 16280MiB |       0%  Default |  
|                           |                           |           N/A |  
+-----+-----+-----+-----+
```

```
+-----+  
| Processes:  
| GPU  GI  CI          PID  Type  Process name          GPU Memory  
| ID   ID          ID   ID          ID          Usage  
+=====+=====+=====+=====+=====+=====+=====+  
| No running processes found  
+-----+
```

```
!zip -r ./output.zip ./output/
```

```
!echonet segmentation --model_name="deeplabv3_resnet50" --num_epochs=200 --output="out  
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWar  
cpuset_checked))  
100% 16/16 [00:56<00:00,  3.56s/it]  
Epoch #0  
100% 12/12 [00:19<00:00,  1.66s/it, 0.5820 (0.2623) / 0.3128 0.1435, 0.0375, 0.038  
100% 65/65 [01:21<00:00,  1.26s/it, 2231.1632 (2963.8004) / 0.3145 0.1385, 0.0000,  
Epoch #1  
100% 12/12 [00:17<00:00,  1.49s/it, 0.2719 (0.2637) / 0.3125 0.1425, 0.0000, 0.000  
100% 65/65 [01:19<00:00,  1.22s/it, 5.5341 (4.0112) / 0.3145 0.1385, 0.0000, 0.000  
Epoch #2  
100% 12/12 [00:17<00:00,  1.45s/it, 0.2099 (0.1661) / 0.3126 0.1425, 0.0540, 0.043  
100% 65/65 [01:21<00:00,  1.25s/it, 0.3834 (0.3948) / 0.3145 0.1385, 0.2831, 0.405  
Epoch #3  
100% 12/12 [00:16<00:00,  1.38s/it, 0.1598 (0.1256) / 0.3132 0.1429, 0.5810, 0.667  
100% 65/65 [01:19<00:00,  1.22s/it, 0.1430 (0.1528) / 0.3145 0.1385, 0.6631, 0.755  
Epoch #4  
100% 12/12 [00:16<00:00,  1.41s/it, 0.1299 (0.1140) / 0.3134 0.1430, 0.6685, 0.735  
100% 65/65 [01:19<00:00,  1.22s/it, 0.1180 (0.1539) / 0.3145 0.1385, 0.6782, 0.745  
Epoch #5  
100% 12/12 [00:17<00:00,  1.46s/it, 0.1122 (0.1031) / 0.3135 0.1424, 0.7153, 0.773  
100% 65/65 [01:21<00:00,  1.25s/it, 0.1048 (0.1187) / 0.3145 0.1385, 0.7389, 0.787  
Epoch #6  
100% 12/12 [00:17<00:00,  1.43s/it, 0.1025 (0.0939) / 0.3121 0.1422, 0.7299, 0.788  
100% 65/65 [01:18<00:00,  1.21s/it, 0.1035 (0.1164) / 0.3145 0.1385, 0.7236, 0.833  
Epoch #7  
100% 12/12 [00:16<00:00,  1.39s/it, 0.0956 (0.0895) / 0.3112 0.1416, 0.7457, 0.817  
100% 65/65 [01:19<00:00,  1.22s/it, 0.0950 (0.0962) / 0.3145 0.1385, 0.7453, 0.790  
Epoch #8  
100% 12/12 [00:17<00:00,  1.42s/it, 0.0891 (0.0802) / 0.3133 0.1435, 0.7684, 0.816  
100% 65/65 [01:21<00:00,  1.26s/it, 0.0890 (0.0915) / 0.3145 0.1385, 0.7679, 0.841  
Epoch #9  
100% 12/12 [00:17<00:00,  1.43s/it, 0.0819 (0.0782) / 0.3123 0.1435, 0.7861, 0.835  
100% 65/65 [01:20<00:00,  1.24s/it, 0.0895 (0.0858) / 0.3145 0.1385, 0.7674, 0.851  
Epoch #10
```

```
Epoch #10
100% 12/12 [00:17<00:00, 1.43s/it, 0.0796 (0.0792) / 0.3106 0.1429, 0.7891, 0.837
100% 65/65 [01:22<00:00, 1.26s/it, 0.0802 (0.1788) / 0.3145 0.1385, 0.7892, 0.838
Epoch #11
100% 12/12 [00:17<00:00, 1.42s/it, 0.0782 (0.0682) / 0.3136 0.1424, 0.7921, 0.849
100% 65/65 [01:19<00:00, 1.22s/it, 0.0785 (0.0627) / 0.3145 0.1385, 0.8004, 0.838
Epoch #12
100% 12/12 [00:16<00:00, 1.38s/it, 0.0740 (0.0719) / 0.3141 0.1432, 0.8057, 0.859
100% 65/65 [01:19<00:00, 1.22s/it, 0.0819 (0.0620) / 0.3145 0.1385, 0.7997, 0.830
Epoch #13
100% 12/12 [00:17<00:00, 1.44s/it, 0.0685 (0.0738) / 0.3122 0.1434, 0.8141, 0.868
100% 65/65 [01:21<00:00, 1.26s/it, 0.0785 (0.0601) / 0.3145 0.1385, 0.7899, 0.833
Epoch #14
100% 12/12 [00:16<00:00, 1.41s/it, 0.0655 (0.0650) / 0.3132 0.1428, 0.8287, 0.870
100% 65/65 [01:19<00:00, 1.22s/it, 0.0765 (0.0694) / 0.3145 0.1385, 0.8085, 0.865
Epoch #15
100% 12/12 [00:16<00:00, 1.39s/it, 0.0625 (0.0627) / 0.3120 0.1420, 0.8325, 0.874
100% 65/65 [01:19<00:00, 1.22s/it, 0.0792 (0.0707) / 0.3145 0.1385, 0.7948, 0.867
Epoch #16
100% 12/12 [00:16<00:00, 1.41s/it, 0.0605 (0.0615) / 0.3112 0.1426, 0.8346, 0.884
100% 65/65 [01:22<00:00, 1.26s/it, 0.0710 (0.0891) / 0.3145 0.1385, 0.8289, 0.864
Epoch #17
100% 12/12 [00:17<00:00, 1.45s/it, 0.0567 (0.0550) / 0.3128 0.1426, 0.8498, 0.888
100% 65/65 [01:20<00:00, 1.24s/it, 0.0675 (0.0523) / 0.3145 0.1385, 0.8240, 0.879
Epoch #18
```

## ▼ Imports

```
#@title Imports

import math
import os
import time
import collections
import pandas

import click
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal
import skimage.draw
import torch
import torchvision
import tqdm
```

Projekt je organiziran u obliku python package-a. Vrhovni modul **echonet** se sastoji od datoteka `__init__.py`, `__main__.py`, `__version__.py`, `config.py` i submodula **datasets** i **utils**.

Submodul **datasets** sadrži datoteke `__init__.py` i `echo.py`, a submodul **utils** `__init__.py`, `segmentation.py` i `video.py`. Cijelu strukturu modula echonet je moguće prikazati na sljedeći način:

- echonet
  - `__init__.py`
  - `main.py`

- \_\_init\_\_.py
- \_\_version\_\_.py
- config.py
- datasets
  - \_\_init\_\_.py
  - echo.py
- utils
  - \_\_init\_\_.py
  - segmentation.py
  - video.py

Datoteka \_\_init\_\_.py se koristi kako bi se naznačilo da se direktorij koristi kao python package. Poziva se pri importanju modula, a obično se koristi za inicijalizaciju packagea. Datoteka \_\_main\_\_.py predstavlja sučelje package-a prema komandnoj liniji, odnosno ta datoteka se automatski izvršava pri pozivu packagea kao skripte iz terminala, dok datoteka \_\_version\_\_.py daje informaciju o trenutnoj verziji.

## ▼ segmentation.py

Datoteka segmentation.py započinje importanjem potrebnih modula među kojima je zanimljiv modul **click**. Ovaj modul omogućava kreiranje sučelja prema komandnoj liniji na brz i jednostavan način. Omogućava ugnježivanje naredbi, učitavanje podnaredbi tijekom izvođenja i automatsko generiranje *help* stranice. Naredba @click.command() definira ime naredbe kojom se skripta u terminalu poziva, @click.option() definira argumente koje je moguće proslijediti skripti prilikom pokretanja. Parametar default određuje vrijednost argumenta kada ona nije definirana pri pozivu, dok parametar type određuje tip vrijednosti koju taj argument prima.

```
import math
import os
import time

import click
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal
import skimage.draw
import torch
import torchvision
import tqdm

import echonet
```

```

@click.command("segmentation")
@click.option("--data_dir", type=click.Path(exists=True, file_okay=False), default=None)
@click.option("--output", type=click.Path(file_okay=False), default=None)
@click.option("--model_name", type=click.Choice(
    sorted(name for name in torchvision.models.segmentation.__dict__
        if name.islower() and not name.startswith("__") and callable(torchvision.mo
    default="deeplabv3_resnet50"))
@click.option("--pretrained/--random", default=False)
@click.option("--weights", type=click.Path(exists=True, dir_okay=False), default=None)
@click.option("--run_test/--skip_test", default=False)
@click.option("--save_video/--skip_video", default=False)
@click.option("--num_epochs", type=int, default=50)
@click.option("--lr", type=float, default=1e-5)
@click.option("--weight_decay", type=float, default=0)
@click.option("--lr_step_period", type=int, default=None)
@click.option("--num_train_patients", type=int, default=None)
@click.option("--num_workers", type=int, default=4)
@click.option("--batch_size", type=int, default=20)
@click.option("--device", type=str, default=None)
@click.option("--seed", type=int, default=0)

```

Metoda **run** obavlja ... Ona prima argumente:

- **data\_dir** koji prima putanju direktorija u kojem se nalazi dataset
- **output** kojom je definirana putanja direktorija u kojem će se spremiti izlazni podaci
- **model\_name** definira model koji će se koristiti za segmentaciju
- **pretrained** određuje hoće li model koristiti pretrenirane težine
- **weights** predstavlja putanju do direktorija u kojem se nalaze inicijalizacijske težine modela
- **run\_test** određuje hoćemo li nad modelom provesti test
- **save\_video** hoćemo li spremiti segmentirani video
- **num\_epochs** definira broj epoha koji će se izvršiti pri treniranju modela
- **lr** definira vrijednost parametra *Learning rate* u algoritmu učenja *Stochastic Gradient Descent*. Taj parametar određuje intenzitet kojim se mijenjaju težine modela u svakom koraku učenja
- **weight\_decay** je dodatni parametar koji definira eksponencijalno opadanje promjena težina prema nuli
- **lr\_step\_period** predstavlja period u kojem dolazi do opadanja parametra *learning rate*, defaultne vrijednosti jednake beskonačnosti
- **num\_workers** je broj podprocesa koji učitavaju podatke
- **batch\_size** broj uzoraka po *batchu*
- **device** definira uređaj na kojem se izvršava skripta.

```

def run(
    data_dir=None,
    output=None,

```

```
model_name="deeplabv3_resnet50",
pretrained=False,
weights=None,

run_test=False,
save_video=False,
num_epochs=50,
lr=1e-5,
weight_decay=1e-5,
lr_step_period=None,
num_train_patients=None,
num_workers=4,
batch_size=20,
device=None,
seed=0,
):
```

Na samom početku metode *run* se resetiraju generatori slučajnih brojeva (postavljanje parametra *seed* na 0). Zatim se metodom *os.path.join()* vrši interakcija sa operacijskim sustavom i stvara jedinstvena izlazna putanja, a metodom *os.makedirs()* se rekursivno stvara cijelo podstablo direktorija prema navedenoj putanji. Parametar *exist\_ok* je postavljen na vrijednost True što označava da se neće prijaviti greška *FileExistsError* ukoliko željeno podstablo već postoji. Sljedećim linijama koda se učitava željeni model iz modula *torchvision* te se vrši postavljanje posljednjeg izlaznog sloja mreže na sloj dvodimenzionalne konvolucije sa jednim izlaznim kanalom.

```
# Seed RNGs
np.random.seed(seed)
torch.manual_seed(seed)

# Set default output directory
if output is None:
    output = os.path.join("output", "segmentation", "{}_{}".format(model_name, "pr
os.makedirs(output, exist_ok=True)

# Set device for computations
if device is None:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Set up model
model = torchvision.models.segmentation.__dict__[model_name](pretrained=pretrained

model.classifier[-1] = torch.nn.Conv2d(model.classifier[-1].in_channels, 1, kernel
```

Ukoliko je to definirano, provjerava li se da li je dostupno postaviti *device* na "cuda", ukoliko nije postavlja se na "cpu". Ako je "cuda", tada se prethodno definirani model paralelizira na najvišem levelu na način da se model dijeli na *chunk-ove* dimenzije definirane *batch\_sizeom* i replicira na svaki dostupni uređaj gdje svaki dio modela prima odgovarajući dio ulaza.

```
if device.type == "cuda":
    model = torch.nn.DataParallel(model)
    model.to(device)
```

Program učitava prethodno spremljene težine modela, ukoliko su dostupne, postavlja parametre za optimiziranje treniranja te pomoću metode `get_mean_and_std` iz submodula `utils` računa statističke parametre (srednje grešku i standardnu devijaciju) podataka za treniranje.

```
if weights is not None:
    checkpoint = torch.load(weights)
    model.load_state_dict(checkpoint['state_dict'])

    # Set up optimizer
    optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=weig
    if lr_step_period is None:
        lr_step_period = math.inf
    scheduler = torch.optim.lr_scheduler.StepLR(optim, lr_step_period)

    # Compute mean and std
    mean, std = echonet.utils.get_mean_and_std(echonet.datasets.Echo(root=data_dir, sp
```

Objekt `kwargs` je *dictionary* u kojem se pohranjuju key-value parovi koji se prosljeđuju kao parametri metodi `echonet.datasets.Echo` koja vraća odgovarajući dataset. Ukoliko je definirani broj pacijenata manji od dužine dataset-a, tada se slučajnim izborom odabire odgovarajući broj uzoraka iz dataseta.

```
tasks = ["LargeFrame", "SmallFrame", "LargeTrace", "SmallTrace"]
kwargs = {"target_type": tasks,
          "mean": mean,
          "std": std
        }

# Set up datasets and dataloaders
dataset = {}
dataset["train"] = echonet.datasets.Echo(root=data_dir, split="train", **kwargs)
if num_train_patients is not None and len(dataset["train"]) > num_train_patients:
    # Subsample patients (used for ablation experiment)
    indices = np.random.choice(len(dataset["train"]), num_train_patients, replace=False)
    dataset["train"] = torch.utils.data.Subset(dataset["train"], indices)
dataset["val"] = echonet.datasets.Echo(root=data_dir, split="val", **kwargs)
```

Iz datoteke `checkpoint.pt`, koja se nalazi u izlaznom direktoriju, se učitavaju parametri modela. Isto tako učitavaju se parametri objekta `optim` koji na temelju sadašnjih parametara modela računa gradijente i nove vrijednosti parametara. Treba napomenuti da je ovim mehanizmom moguće pojedinačno pokretati epohe treniranja i slijedno ih nastavljati jednu na drugu, pri čemu se u `checkpoint["epoch"]` spremi redni broj prethodno obrađene epohe, a u

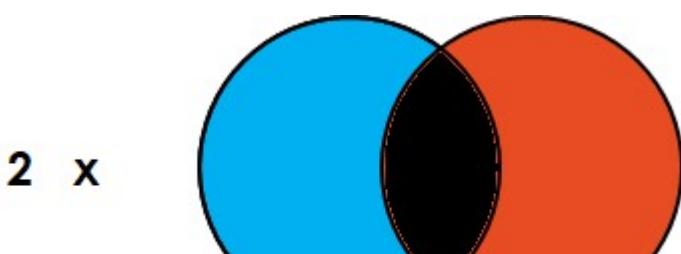
`checkpoint["best_loss"]` najmanja vrijednost gubitka tokom validacije u bilo kojoj dosadašnjoj epohi.

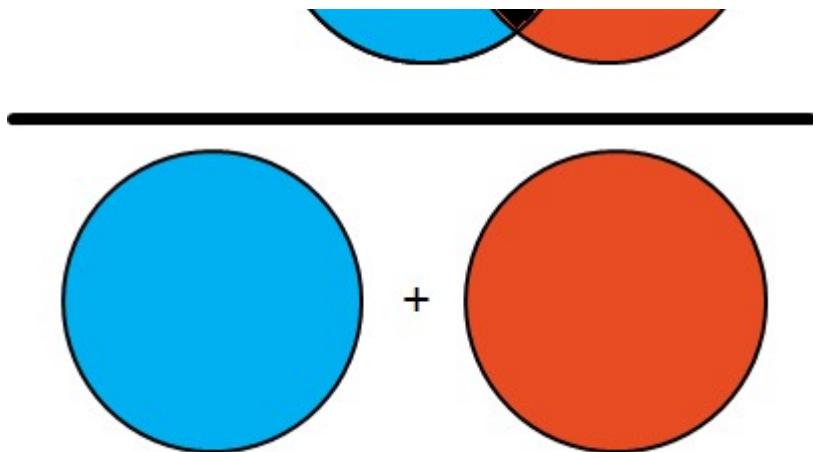
Ukoliko nije moguće učitati sadržaj datoteke `checkpoint.pt`, treniranje modela se pokreće od početne (prve) epohe

```
with open(os.path.join(output, "log.csv"), "a") as f:
    epoch_resume = 0
    bestLoss = float("inf")
    try:
        # Attempt to load checkpoint
        checkpoint = torch.load(os.path.join(output, "checkpoint.pt"))
        model.load_state_dict(checkpoint['state_dict'])
        optim.load_state_dict(checkpoint['opt_dict'])
        scheduler.load_state_dict(checkpoint['scheduler_dict'])
        epoch_resume = checkpoint["epoch"] + 1
        bestLoss = checkpoint["best_loss"]
        f.write("Resuming from epoch {}\n".format(epoch_resume))
    except FileNotFoundError:
        f.write("Starting run from scratch\n")
```

Upotrebom for petlje se nastavlja izvršavanje postupka treniranja od trenutno dostignute epohe do krajnje definirane epohe. U svakoj iteraciji petlje se provodi treniranje i validacija modela. Početno odgovarajući dio dataset-a pošalje kao parametar konstruktoru objekta `dataloader` klase `torch.utils.data.DataLoader` čime se omogućuje jednostavan obilazak podataka, što je i detaljno objašnjeno u nastavku teksta. Objekt `dataloader` se kao parametar proslijeđuje metodi `run_epoch`. Na temelju njenih povratnih vrijednosti se računaju parametri ***large\_dice***, ***small\_dice***, ***overall\_dice***. Sasvim općenito, *Dice Coefficient* je parametar koji služi za metriku modela koji obavlja semantičku segmentaciju. Definira se kao postotak piksela na slici koji su ispravno segmentirani. Promotrimo li primjer na donjoj slici, jasno je da se računa kao omjer dvostrukog broja preklapajućih piksela segmentirane slike i njene maske i ukupnog broja piksela na maski i segmentiranoj slici koji su obuhvaćeni unutar traženog oblika. U našem slučaju definiramo tri *Dice Coefficient*-a:

- ***overall\_dice***: Definira se aritmetički prosjek *Dice Coefficient*-a za sve ehokardiograme u `dataset-u` i to za `frame-ove` na kojima je srce u sistoli i za one na kojima je srce u dijastoli
- ***large\_dice***: Definira se kao aritmetički prosjek *Dice Coefficient*-a za sve ehokardiograme i to za `frame-ove` na kojima je srce u dijastoli
- ***small\_dice***: Definira se kao aritmetički prosjek *Dice Coefficient*-a za sve ehokardiograme i to za `frame-ove` na kojima je srce u sistoli





Nakon završetka izvršavanja metode `run_epoch` ispisujemo rezultate obrade u datoteku `log.csv` u izlaznom direktoriju. Tu su prikazani razni parametri poput rednog broja epohe, maksimalne alocirane memorije, parametara `overall_dice`, `small_dice`, vremena izvršavanja itd...

```
for epoch in range(epoch_resume, num_epochs):
    print("Epoch #{}".format(epoch), flush=True)
    for phase in ['train', 'val']:
        start_time = time.time()
        for i in range(torch.cuda.device_count()):
            torch.cuda.reset_peak_memory_stats(i)

        ds = dataset[phase]
        dataloader = torch.utils.data.DataLoader(
            ds, batch_size=batch_size, num_workers=num_workers, shuffle=True,
            loss, large_inter, large_union, small_inter, small_union = echonet.uti
            overall_dice = 2 * (large_inter.sum() + small_inter.sum()) / (large_un
            large_dice = 2 * large_inter.sum() / (large_union.sum() + large_inter.
            small_dice = 2 * small_inter.sum() / (small_union.sum() + small_inter.
            f.write("{}\n".format(epoch,
                                  phase,
                                  loss,
                                  overall_dice,
                                  large_dice,
                                  small_dice,
                                  time.time() - star
                                  large_inter.size,
                                  sum(torch.cuda.max
                                  sum(torch.cuda.max
                                  batch_size))

            f.flush()
            scheduler.step()
```

U sljedećem koraku vrši se spremanje dosadašnjih rezultata obrade u objekt `save`, tipa `dictionary`, koji se spremi u izlazni direktorij pod nazivom "checkpoint.pt". Ukoliko je gubitak učravo obrađene epohe manji od naimanied dosadašnied zabilježenoa aubitka. `dictionary`

save se spremu u datoteku *best.pt*.

```
# Save checkpoint
    save = {
        'epoch': epoch,
        'state_dict': model.state_dict(),
        'best_loss': bestLoss,
        'loss': loss,
        'opt_dict': optim.state_dict(),
        'scheduler_dict': scheduler.state_dict(),
    }
    torch.save(save, os.path.join(output, "checkpoint.pt"))
    if loss < bestLoss:
        torch.save(save, os.path.join(output, "best.pt"))
        bestLoss = loss
```

Nakon obrade svih epoha model učitava parametre modela iz epohe sa najmanjim gubitkom. Ukoliko je argument *run\_test* zadan kao True, izvršava se testiranje modela. Metodom *echonet.datasets.Echo* se učitava dio dataseta predviđen za testiranje i validaciju. Ponovno se formira objekt dataloader koji omogućuje lakše rukovanje i proslijedivanje podataka modelu, a zatim se metodom *run\_epoch* pokreće testiranje i validacija. Provodi se proračun već objašnjenih parametara *overall\_dice*, *small\_dice*, *large\_dice*. Razlika je što su to ovog puta jednodimenzionalne liste, duljine jednakе broju uzoraka dataseta. Dakle, za svaki uzorak dataset-a, odnosno svaki ehokardiogram se računaju zasebni parametri spremljeni kao pojedinačni element liste i potom spremaju u odgovarajuće datoteke *val\_dice.csv*, odnosno *test\_dice.csv*.

```
if run_test:
    # Run on validation and test
    for split in ["val", "test"]:
        dataset = echonet.datasets.Echo(root=data_dir, split=split, **kwargs)
        dataloader = torch.utils.data.DataLoader(dataset,
                                                batch_size=batch_size, num_workers=0)
        loss, large_inter, large_union, small_inter, small_union = echonet.utils.dice(
            dataloader)

        overall_dice = 2 * (large_inter + small_inter) / (large_union + large_inter)
        large_dice = 2 * large_inter / (large_union + large_inter)
        small_dice = 2 * small_inter / (small_union + small_inter)

        with open(os.path.join(output, "{}_dice.csv".format(split)), "w") as g:
            g.write("Filename, Overall, Large, Small\n")
            for (filename, overall, large, small) in zip(dataset.fnames, overall_dice,
                                                          large_dice, small_dice):
                g.write("{} , {}, {}, {} \n".format(filename, overall, large, small))

            f.write("{} dice (overall): {:.4f} ({:.4f} - {:.4f})\n".format(split,
                overall_dice, overall_dice - large_dice))
            f.write("{} dice (large): {:.4f} ({:.4f} - {:.4f})\n".format(split,
                large_dice, large_dice - small_dice))
            f.write("{} dice (small): {:.4f} ({:.4f} - {:.4f})\n".format(split,
                small_dice, small_dice - overall_dice))
            f.flush()
```

U idućem koraku, ukoliko parametar `save_video` ima vrijednost `True` i ukoliko u izlaznom direktoriju u poddirektoriju `videos` ne postoje sve datoteke imena iz liste `dataloader.dataset.fnames`, odnosno svi videi iz testnog dijela `dataset-a`, pokreće se blok koda koji ima sljedeće funkcionalnosti.

Metoda `model.eval()` isključuje određene slojeve modela koji se ponašaju drugačije za vrijeme treniranja i evaluacije. To bi primjerice bili `DropOut` i `BatchNorm` slojevi.

Metodom `torch.no_grad()` se mijenja kontekst modela čime se isključuju algoritmi za gradijentne proračune. Ova metoda rezultira uštedom u potrošnji memorije prilikom proračuna.

Obilazimo sve uzorke (ehokardiograme) u testnom dataset-u i pomoću metode `tqdm.tqdm(dataloader)` pratimo napredak na *progress bar-u*.

Pošto, bi segmentacija cijelog videa bila riskantna zbog potencijalne prevelike duljine videa, video se segmentira u manjim dijelovima, duljine (brojem `frame-ova`) određene parametrom `batch_size`. Takvi segmentirani odsječci se naponsljetu konkataniraju u jedan cjelovit segmentirani video.

U sljedećem koraku za svaki ehokardiogram imamo par originalnog videa koji je reprezentiran tenzorom `video` i izlaznog videa iz modela zadanog tenzorom `logit`. Nakon što denormaliziramo `video`, odnosno pomnožimo svaki piksel videa sa standardnom devijacijom, a zatim na tu vrijednost dodamo srednju grešku, konkateniramo dva identična tenzora videa jedan pored drugoga, odnosno po osi 3 (odgovara osi koja određuje širinu frame-ova videa). 'Ljeva' kopija videa će biti originalni video, dok će 'desna' kopija predstavljati segmentirani video. Segmentirani video se dobija na način da se za one piksele tenzora `logit` koji imaju vrijednost 1, u 'desnoj' kopiji tenzora `video` u plavom kanalu (kanal 0, pošto radimo sa BGR modelom) postavlja vrijednost 255, a za one piksele tenzora `logit` koji imaju vrijednost 0 se postavlja vrijednost 0.

```
# Save videos with segmentation
if save_video and not all(os.path.isfile(os.path.join(output, "videos", f)) for f
    # Only run if missing videos

    model.eval()

    os.makedirs(os.path.join(output, "videos"), exist_ok=True)
    os.makedirs(os.path.join(output, "size"), exist_ok=True)
    echonet.utils.latexify()

    with torch.no_grad():
        with open(os.path.join(output, "size.csv"), "w") as g:
            g.write("Filename,Frame,Size,HumanLarge,HumanSmall,ComputerSmall\n")
            for (x, (filenames, large_index, small_index), length) in tqdm.tqdm(da
                # Run segmentation model on blocks of frames one-by-one
                # The whole concatenated video may be too long to run together
                y = np.concatenate([model(x[i:(i + batch_size), :, :, :].to(device
```

```

start = 0
x = x.numpy()
for (i, (filename, offset)) in enumerate(zip(filenames, length)):
    # Extract one video and segmentation predictions
    video = x[start:(start + offset), ...]
    logit = y[start:(start + offset), 0, :, :]

    # Un-normalize video
    video *= std.reshape(1, 3, 1, 1)
    video += mean.reshape(1, 3, 1, 1)

    # Get frames, channels, height, and width
    f, c, h, w = video.shape # pylint: disable=W0612
    assert c == 3

    # Put two copies of the video side by side
    video = np.concatenate((video, video), 3)

    # If a pixel is in the segmentation, saturate blue channel
    # Leave alone otherwise
    video[:, 0, :, w:] = np.maximum(255. * (logit > 0), video[:, 0,

```

U idućim koracima računamo segmentirani broj piksela po svakom *frame-u*, odnosno segmentiranu površinu po *frame-u* i zapisujemo te vrijenosti u jednodimenzionalni tenzor *size*. Iz sortiranog tenzora nalazimo dvije vrijednosti *trim\_min* i *trim\_max*. One predstavljaju minimalnu, odnosno maksimalnu segmentiranu površinu, a dobijene su kao vrijednosti koje se nalaze na indeksu  $0.05/\text{len}(\text{size})$ , odnosno indeksu  $0.95\text{len}(\text{size})$  sortiranog tenzora *size*. Pomoću metode *scipy.signal.find\_peaks* pronađemo lokalne maksimume tenzora *size*. Pošto je unutar same metode proslijedena invertirana (negirana) verzija tenzora *size*, tada lokalni maksimumi, u biti odgovaraju segmentiranoj površini *frame-ova* koji prikazuju srce u položaju sistole. Ova metoda zaista vraća istinite vrijednosti, pošto su njeni parametri *distance* i *prominence* postavljeni na 20 odnosno na polovicu razlike maksimalne i minimalne segmentirane površine. *Distance* definira minimalni razmak dva framea na kojem je detektirana sistola, a *prominence* definira relativnu granicu iznad koje se detektira lokalni maksimum.

```

# Compute size of segmentation per frame
size = (logit > 0).sum((1, 2))

# Identify systole frames with peak detection
trim_min = sorted(size)[round(len(size) ** 0.05)]
trim_max = sorted(size)[round(len(size) ** 0.95)]
trim_range = trim_max - trim_min
systole = set(scipy.signal.find_peaks(-size, distance=20, prom

# Write sizes and frames to file
for (frame, s) in enumerate(size):
    s.write("{}.\n".format(filename, frame, s))

```

Sljedeći dijelovi koda služe za grafički prikazivanje rezultata segmentacije. Naime, prikazujemo vremensku ovisnost površine lijeve ventrikule uz korištenje modula *matplotlib.pyplot*. Prvo, iscrtavamo scatter graf koji se sastoji od vrijednosti iz tenzora *size*, a zatim uz pomoć metode *plot* iscrtavamo vertikalne ravne linije u vremenskim trenucima koji odgovaraju položaju sistole, odnosno minimalne površine lijeve ventrikule srca. Ovaj postupak je popraćen i dodatnim kodom koji se odnosi na uređivanje i labeliranje samog grafa.

```
# Plot sizes
fig = plt.figure(figsize=(size.shape[0] / 50 * 1.5, 3))
plt.scatter(np.arange(size.shape[0]) / 50, size, s=1)
ylim = plt.ylim()
for s in systole:
    plt.plot(np.array([s, s]) / 50, ylim, linewidth=1)
plt.ylim(ylim)
plt.title(os.path.splitext(filename)[0])
plt.xlabel("Seconds")
plt.ylabel("Size (pixels)")
plt.tight_layout()
plt.savefig(os.path.join(output, "size", os.path.splitext(file
plt.close(fig)
```

Na samom početku se jednodimenzionalni tenzor *size* normalizira. Ukoliko se od vrijednosti svakog elementa tenzora oduzme minimalna vrijednost, a zatim podijeli sa maksimalnom vrijednosti takvog umanjenog tenzora, dobije se tenzor čije su vrijednosti unutar intervala [ 0 , 1 ]. U nastavku, iteriramo kroz elemente niza *size*, i na svakom *frame* videa bijelom bojom označavamo piksel videa koji je povezan sa vrijednosti trenutnog elementa niza *size*, što rezultira tankom horizontalnom bijelom linijom. Sličan postupak se obavlja i ukoliko je trenutni element, jedan od *frame*ova zabilježen u listi *systole*.

Metoda *dash* stvara listu indeksa prema sljedećem pravilu. Ukoliko *dash* ima sljedeći poziv:  
*dash(10,30,on=2,off=5)*

,tada je povratna vrijednost metode lista [10,11,17,18,24,25], odnosno listu extenda sa onoliko brojeva u nizu koliki je parametar *on*, krenuvši napočetku od broja *start*, zatim preskače *off* vrijednosti, ponovno dodaje *on* vrijednosti i tako u krug dok ne dostigne vrijednost definiranu parametrom *stop*. Ova lista služi za indeksiranje piksela koji obojiti u zeleno, ukoliko je trenutni *frame* od strane vanjskih ekperata označen kao *frame* u kojem je srce u položaju dijastole. Isto tako pikseli indeksirani na temelju ove liste će se obojiti u crveno ukoliko je trenutni *frame* označen kao *frame* u kojem se srce nalazi u položaju sistole. Ovakvo označavanje će rezultirati nizom jednakih udaljenih, paralelnih o relativno debelih vertikalnih linija.

Nadalje, iscrtavamo bijeli krug sa središtem određenim indeksom trenutnog *framea* i

površinom lijeve ventrikule koja se nalazi na njemu, radiusa 4. i incna.

```
# Normalize size to [0, 1]
    size -= size.min()
    size = size / size.max()
    size = 1 - size

    # Iterate the frames in this video
    for (f, s) in enumerate(size):

        # On all frames, mark a pixel for the size of the frame
        video[:, :, int(round(115 + 100 * s)), int(round(f / len(size) * 200 + 10))] = size

        if f in systole:
            # If frame is computer-selected systole, mark with a 1
            video[:, :, 115:224, int(round(f / len(size) * 200 + 10))] = 1

    def dash(start, stop, on=10, off=10):
        buf = []
        x = start
        while x < stop:
            buf.extend(range(x, x + on))
            x += on
            x += off
        buf = np.array(buf)
        buf = buf[buf < stop]
        return buf
    d = dash(115, 224)

    if f == large_index[i]:
        # If frame is human-selected diastole, mark with green
        video[:, :, d, int(round(f / len(size) * 200 + 10))] = 0
    if f == small_index[i]:
        # If frame is human-selected systole, mark with red da
        video[:, :, d, int(round(f / len(size) * 200 + 10))] = 1

    # Get pixels for a circle centered on the pixel
    r, c = skimage.draw.disk((int(round(115 + 100 * s)), int(round(f / len(size) * 200 + 10))), 4)

    # On the frame that's being shown, put a circle over the pixel
    video[f, :, r, c] = 255.
```

Na samom kraju, segmentirani video transponiramo. Posjetimo on je do ovog trenutka bio tenzor oblika [f, c, h, w], a nakon transponiranja ima oblik [c, f, h, w]. Takav video spremamo u poddirektorij *videos* izlaznog direktorija (sa ekstenzijom .avi).

```
# Rearrange dimensions and save
    video = video.transpose(1, 0, 2, 3)
    video = video.astype(np.uint8)
    echonet.utils.savevideo(os.path.join(output, "videos", filename), video)
```

*Metoda run\_epoch()* predstavlja srž ovog programa, no što je to metoda koja služi za trening?

metoda `run_epoch()` predstavlja se u ovom programu, posluje je to metoda koja služi za treniranje i evaluaciju modela. Prima argumente:

- **model**: model nad kojim je potrebno provesti treniranje ili evaluaciju
- **dataloader**: objekt klase `torch.utils.data.DataLoader` koji služi za iteraciju kroz dataset
- **optim**: objekt klase `torch.optim.SGD`, već prethodno objašnjen - **device**: objekt kojim se naznačava uređaj ("cpu" ili "cuda") na kojem se vrše proračuni.

Povratne vrijednosti ove metode su:

- **total/n/112/112**: srednja vrijednost gubitka segmentacije (ovaj parametar, kao i metoda kojom se računa će biti detaljno objašnjeni u nastavku)
- **large\_inter\_list**: lista čiji elementi predstavlja broj piksela u kojem se preklapaju segmentirana ventrikula nastala kao izlaz iz modela i ona koju je obilježio vanjski ekspert, definirana kao maska u klasi *Echo* submodula *datasets*, za srce koje je trenutno u dijastoli
- **large\_union\_list** lista čiji elementi predstavljaju zbroj piksela lijeve ventrikule na segmentiranom *frameu* i broj piksela ventrikule definirane kao maska u klasi *Echo* submodula *dataset-a*, za srce koje je trenutno u dijastoli
- **small\_inter\_list**: lista čiji elementi predstavlja broj piksela u kojem se preklapaju segmentirana ventrikula nastala kao izlaz iz modela i ona koju je obilježio vanjski ekspert, definirana kao maska u klasi *Echo* submodula *datasets*, za srce koje je trenutno u sistoli
- **small\_union\_list** lista čiji elementi predstavljaju zbroj piksela lijeve ventrikule na segmentiranom *frameu* i broj piksela ventrikule definirane kao maska u klasi *Echo* submodula *dataset-a*, za srce koje je trenutno u sistoli

```
def run_epoch(model, dataloader, train, optim, device):
    """Run one epoch of training/evaluation for segmentation.

    Args:
        model (torch.nn.Module): Model to train/evaluate.
        dataloder (torch.utils.data.DataLoader): Dataloader for dataset.
        train (bool): Whether or not to train model.
        optim (torch.optim.Optimizer): Optimizer
        device (torch.device): Device to run on
    """

```

Double-click (or enter) to edit

U nastavku se računa broj piksela koji se nalaze unutar područja ventrikule označenog od strane eksperta za sistolu i za dijastolu te se taj broj spremi u varijablu *pos*. Slično se radi i za varijablu *neg*, no ovaj put je to broj piksela izvan segmentiranog područja.

Isti ovaj postupak se ponavlja još jednom, međutim ovaj put se vrijednosti prebacuju u memoriju *cpua* u odgovarajuće varijable *pos\_pix* i *neg\_pix*.

```

with torch.set_grad_enabled(train):
    with tqdm.tqdm(total=len(dataloader)) as pbar:
        for (_, (large_frame, small_frame, large_trace, small_trace)) in dataloade
            # Count number of pixels in/out of human segmentation
            pos += (large_trace == 1).sum().item()
            pos += (small_trace == 1).sum().item()
            neg += (large_trace == 0).sum().item()
            neg += (small_trace == 0).sum().item()

            # Count number of pixels in/out of computer segmentation
            pos_pix += (large_trace == 1).sum(0).to("cpu").detach().numpy()
            pos_pix += (small_trace == 1).sum(0).to("cpu").detach().numpy()
            neg_pix += (large_trace == 0).sum(0).to("cpu").detach().numpy()
            neg_pix += (small_trace == 0).sum(0).to("cpu").detach().numpy()

```

Tenzori *large\_frame* i *large\_trace* se prebacuju u memoriju izabranog uređaja i svi proračuni koji će biti spomenuti u nastavku vrše se na tom uređaju. Dakle *large\_frame* proslijedujemo kao ulaz modelu i za njega dobijamo izlazni segmentirani *frame* *y\_large*. Metodom *binary\_cross\_entropy\_with\_logits* iz modula *torch.nn.functional* računamo gubitak segmentacije. Ova metoda računa gubitak prema sljedećoj formuli:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

Nadalje, određujemo broj piksela u kojem se preklapaju segmentirani dijelovi *frameova* *large\_trace* i *y\_large*. Tu vrijednost pohranjujemo u varijablu *large\_inter*. Isto tako određujemo i broj piksela koji nastaje unijom ova dva *framea* i to pohranjujemo u varijablu *large\_union*. Navedene vrijednosti spremamo u liste *large\_inter\_list* i *large\_union\_list*. Sav ovaj postupak ponavljamo i za *frame* u kojem je srce u sistoli i na taj način dobijamo liste *small\_inter\_list* i *small\_union\_list*.

```

# Run prediction for diastolic frames and compute loss
    large_frame = large_frame.to(device)
    large_trace = large_trace.to(device)
    y_large = model(large_frame)[ "out"]
    loss_large = torch.nn.functional.binary_cross_entropy_with_logits(y_la
    # Compute pixel intersection and union between human and computer segm
    large_inter += np.logical_and(y_large[:, 0, :, :].detach().cpu().numpy(
    large_union += np.logical_or(y_large[:, 0, :, :].detach().cpu().numpy(
    large_inter_list.extend(np.logical_and(y_large[:, 0, :, :].detach().cp
    large_union_list.extend(np.logical_or(y_large[:, 0, :, :].detach().cpu

    # Run prediction for systolic frames and compute loss
    small_frame = small_frame.to(device)
    small_trace = small_trace.to(device)
    y_small = model(small_frame)[ "out"]
    loss_small = torch.nn.functional.binary_cross_entropy_with_logits(y_sm

```

```
# Compute pixel intersection and union between human and computer segm
small_inter += np.logical_and(y_small[:, 0, :, :].detach().cpu().numpy()
small_union += np.logical_or(y_small[:, 0, :, :].detach().cpu().numpy()
small_inter_list.extend(np.logical_and(y_small[:, 0, :, :].detach().cp
small_union_list.extend(np.logical_or(y_small[:, 0, :, :].detach().cpu
```

Ukoliko smo ovu metodu pozvali za treniranje , a ne validaciju izvršavmo metodu `pytorch.optim.Optimizer.zero_grad()` koja ručno postavlja sve gradijente na nulu prije nego što izvršimo unazadnu propagaciju. Ovo je potrebno izvršiti jer PyTorch akumulira vrijednosti gradijenata pri uzastopnim unazadnim propagacijama, a to je oželjno jedino kada želimo akumulirati gubitak više batcheva.

`loss.backward()` je metoda koja računa parcijalnu derivaciju gubitka po svakoj od težine. Možemo to zapisati u sljedećem obliku:

`x.grad += dloss/dx` , `x` predstavlja neku proizvoljnu težinu modela

`optim.step()` sada računa nove vrijednosti težina prema sljedećem zakonu:

`x += -lr * x.grad, lr` predstavlja *learning rate*

Sada je poprilično intuitivno zašto pozivamo metodu `zero_grad`, jer da je ne pozivamo gubici više faza bi se nakupili u varijabli `x.grad` i ne bismo imali ispravnu derivaciju, odnosno novu vrijednost težine.

```
loss = (loss_large + loss_small) / 2
      if train:
          optim.zero_grad()
          loss.backward()
          optim.step()
```

Na samom kraju metode, vršimo proračun parametara  $p$  i  $p_{pix}$  tako da oni predstavljaju udjel segmentiranih piksela u ukupnom broju piksela dotičnog *framea*. Prethodno objašnjene liste `large_inter_list`, `large_union_list`, `small_inter_list` i `small_union_list` se formatiraju u *numpy array* i takve vraćaju kao povratne vrijednosti metode.

```
# Accumulate losses and compute baselines
    total += loss.item()
    n += large_trace.size(0)
    p = pos / (pos + neg)
    p_pix = (pos_pix + 1) / (pos_pix + neg_pix + 2)

    # Show info on process bar
    pbar.set_postfix_str("{:.4f} {:.4f} / {:.4f} {:.4f}, {:.4f}, {:.4f}"
    pbar.update()

large_inter_list = np.array(large_inter_list)
large_union_list = np.array(large_union_list)
small_inter_list = np.array(small_inter_list)
small_union_list = np.array(small_union_list)
```

```

    small_inter_list = np.array(small_inter_list)

    return (total / n / 112 / 112,
            large_inter_list,
            large_union_list,
            small_inter_list,
            small_union_list,
            )

```

## ❖ utils/\_\_init\_\_.py

### ❖ loadvideo()

Metoda za argument prima putanju do videa koji je potrebno učitati. Učitavanje se realizira korištenjem cv2 instance **OpenCV 2** modula.

Prvo se putem `cv2.VideoCapture` metode definira video objekt iz kojeg se izvlače informacije o rezoluciji( $width \times height$ ) i broju *frame*-ova videa. Podaci za video se zatim učitavaju *frame-by-frame* i to na način da se za svaki *frame* vrate R,G,B vrijednosti u obliku niza od 3 8-bitna integera(vrijednosti 0-255) za svaki njegov piksel.

Nakon učitavanja podataka za sve *frame*-ove videa rezultat je tenzor čiji je format `[frame][X][Y] = [R,G,B]` odnosno `[frame, height, width, channels]`. Za potrebe našeg računanja želimo `[channels, frame, height, width]` format što postižemo korištenjem **NumPy transpose()** metode koja mijenja položaj osi tenzora.

```

def loadvideo(filename: str) -> np.ndarray:

    if not os.path.exists(filename):
        raise FileNotFoundError(filename)
    # Kreiraj video objekt
    capture = cv2.VideoCapture(filename)

    frame_count = int(capture.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_width = int(capture.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(capture.get(cv2.CAP_PROP_FRAME_HEIGHT))
    # Tensor [frame, height, width, channels] formata -> elementi tenzora su 3-člani niz
    v = np.zeros((frame_count, frame_height, frame_width, 3), np.uint8)

    for count in range(frame_count):
        # Frame-by-frame čitanje
        ret, frame = capture.read()
        # ret označava je li frame ispravno pročitan
        # Frame je matrica s 3-članim nizovima kao elementima
        if not ret:
            raise ValueError("Failed to load frame #{} of {}".format(count, filename))
        # OpenCV koristi BGR format po defaultu
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

```

```

# Za zadani frame postavi pročitane R,G,B vrijednosti za svaki piksel
v[count, :, :] = frame

# Transpose mehanizam -> treća os trenutnog tenzora postaje prva os novog tenzora, p
# Prije: [frame, height, width, channels]
v = v.transpose((3, 0, 1, 2))
# Nakon: [channels, frame, height, width]

return v

```

## ▼ savevideo()

Budući da smo za učitavanje videa koristili **OpenCV 2** modul isti se koristi i za spremanje videa. Spremanje videa može biti korisno kada želimo spremiti rezultate segmentacije videa.

Za potrebe spremanja videa koristimo objekt `cv2.VideoWriter` klase kojoj specificiramo:

- putanju koja uključuje ime spremljenog videa i definira lokaciju spremanja
- 4 bitni kod koji definira način enkodiranja i kompresiranja videa poznatiji pod oznakom `fourcc`. U našem slučaju koristimo `MJPEG` - Motion JPEG
- `FPS(frames per seconds)` videa kojeg spremamo
- rezolucija videa odnosno dimenzije `frame-a` u obliku **Python tuple-a**

Nakon kreiranja `VideoWriter` instance pohranujemo video *frame-by-frame* pri čemu video podatke koji su dimenzije [channels, frames, height, width] moramo pretvoriti u [frame, height, width, channels] korištenjem **NumPy transpose** metode.

Također, prilikom učitavanja videa u `loadvideo()` metodi smo učitane *frame-ove* pretvarali iz BGR u RGB *color space* budući da **OpenCV** koristi BGR format po *defaultu*. Stoga zaključujemo da ćemo prilikom spremanja videa morati napraviti pretvorbu iz RGB u BGR *color space*.

```

def savevideo(filename: str, array: np.ndarray, fps: typing.Union[float, int] = 1):
    c, _, height, width = array.shape

    if c != 3:
        raise ValueError("savevideo expects array of shape (channels=3, frames, height, width)")
    fourcc = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
    out = cv2.VideoWriter(filename, fourcc, fps, (width, height))

    # pretvori u [frame, height, width, channels] dimenzije
    for frame in array.transpose((1, 2, 3, 0)):
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
        out.write(frame)

```

## ▼ get\_mean\_and\_std()

Za potrebe normalizacije R,G,B vrijednosti potrebne su nam vrijednosti aritmetičke sredine i standardne devijacije koje ćemo izračunati zadanom metodom.

Metoda računa aritmetičku sredinu i standardnu devijaciju za svaki R,G,B kanal posebno odnosno vraća **Python tuple** koji se sastoji od 3-članih nizova aritmetičke sredine i standardne devijacije pri čemu svaki član predstavlja navedeni kvantifikator za R,G,B kanal.

Koristimo formule za negrupirane podatke:

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2 - \bar{X}}$$

Metoda za argumente prima:

- **dataset** : **PyTorch dataset** objekt koji kreiramo prije poziva metode s **echo.py** modulom. Za svaku instancu imamo video varijablu koja sadrži R,G,B podatke grupirane po frame-ovima i target varijablu koja nam u ovom slučaju nije potrebna.
- **samples** : broj instanci **dataset-a** koje ćemo uzeti u obzir prilikom računanja zadanih statističkih kvantifikatora. Ako je vrijednost `None`, za računanje se koriste sve instance **dataset-a**. U suprotnom na slučajan način biramo **samples** instanci iz cijelog **dataset-a**.
- **batch\_size** : parametar za **dataloader** prilikom učitavanja podataka. Detaljnije obrađeno u poglavlju **PyTorch Dataloader**
- **num\_workers** : parametar za **dataloader** prilikom učitavanja podataka. Detaljnije obrađeno u poglavlju **PyTorch Dataloader**

Format parametra koji nam vraća **dataloader** u iteraciji je tenzor čija prva dimenzija predstavlja broj videa koji se nalaze unutar batch-a, a zatim svaki video slijedi [channel, frame, height, width] format. Budući da nam je za računanje aritmetičke sredine i standardne devijacije potrebna suma R,G,B vrijednosti naš cilj je manipulacijom dimenzija tenzora omogućiti zbrajanje zasebno po svakom kanalu.

Kako bi postigli naš cilj dobiveni tenzor trebamo transponirati u 2D oblik u kojem će prva dimenzija biti dimenzija R,G,B kanala dok će druga dimenzija/stupci predstavljati R,G,B vrijednosti svakog piksela među svim frame-ovima od svih videa unutar batch-a. Kada postignemo takav oblik, traženu sumu i kvadratnu sumu možemo jednostavno postići zbrajanjem po stupcima.

Stoga u prvom koraku korištenjem **PyTorch transpose** metode transponiramo **dataloader** tenzor čime naša prva dimenzija postaje dimenzija R,G,B kanala.

Korištenjem **view** metode definiramo dimenzije/format u koji želimo pretvoriti trenutni **dataloader** tenzor. Kao što smo prethodno spomenuli, želimo da nam prva dimenzija bude dimenzija R,G,B kanala(što smo upravo postigli s transponiranjem) dok unutar druge dimenzije želimo *flatt-ati* piksele svih frame-ova i videa u jednu dimenziju.

To postižemo na način da `view` metodi za prvi parametar proslijedimo veličinu prve dimenzije(u našem slučaju 3) dok za drugi parametar proslijedimo -1 što će **PyTorch-u** "reći" da veličina druge dimenzije može biti bilo koji broj koji će nam omogućiti da u 3 retka postavimo podatke o R,G,B vrijednostima svih piksela od svih *frame-ova* i videa unutar *batch-a*.

Znamo da će to biti moguće jedino ukoliko broj stupaca bude jednak broju svih piksela pri čemu će njihove R,G,B vrijednosti biti posložene u R,G,B retcima. Na ovaj način veličina druge dimenzije predstavlja ukupan broj sumiranih piksela koji nam je također bitan za izračun aritmetičke sredine i standardne devijacije.

Korištenjem *contiguous* metode **PyTorch** će za pohranu našeg novog 2D tenzora alocirati kontinuirane memoriske lokacije čime će se znatno ubrzati postupak zbrajanja po retcima zbog jednostavnog i brzog adresiranja podataka što je zorno prikazano na donje 2 slike.

Također, određene **PyTorch** operacije zahtijevaju da su njihovi ulazni parametri *contiguous*. Jedna od takvih metoda je i `view` metoda. Budući da će prethodno transponiranje tenzora uzrokovati *non-contiguous* format, prije primjene `view` metode potrebno je transponirani tenzor pretvoriti u *contiguous* format.

#### Standardni format 2D niza/matrice:

0	1	2	3
4	5	6	7
8	9	10	11

#### Kontinuirani format 2D niza/matrice:

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

```
def get_mean_and_std(dataset: torch.utils.data.Dataset,
                     samples: int = 128,
                     batch_size: int = 8,
                     num_workers: int = 4):

    if samples is not None and len(dataset) > samples:
        indices = np.random.choice(len(dataset), samples, replace=False)
        dataset = torch.utils.data.Subset(dataset, indices)
    dataloader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, num_workers=num_workers, shuffle=True)

    n = 0 # broj obrađenih elemenata
    s1 = 0. # suma R,G,B vrijednosti odvojeno za svaki kanal. 3-člani niz
    s2 = 0. # kvadratna suma R,G,B vrijednosti odvojeno za svaki kanal. 3-člani niz
    for (x, _) in tqdm.tqdm(dataloader):
```

```
# Kreiraj 2D tenzor čija prva dimenzija odgovara R,G,B kanalima, a druga dimen
x = x.transpose(0, 1).contiguous().view(3, -1)
# Broj sumiranih piksela = broj stupaca
n += x.shape[1]
# Obična suma po stupcima(dim=1) -> rezultat je 3 člani niz s vrijednostima za
s1 += torch.sum(x, dim=1).numpy()
# Kvadratna suma po stupcima(dim=1) -> rezultat je 3 člani niz s vrijednostima
s2 += torch.sum(x ** 2, dim=1).numpy()
# Izračun parametara po formulama
mean = s1 / n
std = np.sqrt(s2 / n - mean ** 2)

mean = mean.astype(np.float32)
std = std.astype(np.float32)

return mean, std
```

## ▼ bootstrap()

Prilikom ispisivanja performansi našeg modela koristimo određene statističke parametre poput **Sørensen–Dice koeficijenta** koji predstavlja jako bitnu mjeru uspješnosti našeg modela.

Srž inferencijalne statistike jest u mogućnosti predviđanja/procjene vrijednosti određenog parametra(npr. aritmetičke sredine, proporcije) u cijeloj populaciji na temelju podataka prisutnih u uzorku uz definiranu razinu sigurnosti koja je predstavljena parametrom signifikantnosti  $\alpha$ .

Vrijednost  $\alpha$  se najčešće postavlja na 5%, pa se stoga naša procjena parametra populacije sastoji u definiranju intervala vrijednosti koji će u 95% slučajeva sadržavati vrijednost parametra cijele populacije. Ovaj interval je poznatiji pod nazivom **interval povjerenja**(*Confidence interval*).

Za izračun intervala povjerenja postoje različite metode sa standardiziranim formulama, ali one zahtjevaju ispunjenje različitih uvjeta. Takve metode nazivamo **parametrijskim** metodama.

Metode koje ne zahtijevaju nikakve pretpostavke/uvjete su **neparametrijske** metode. Jedna od neparametrijskih tehnika za izračunavanje intervala povjerenja je tzv. **bootstrap** tehnika.

Bootstrap tehnika počiva na *CLT(Central Limit Theorem)*. Naime, interval povjerenja se određuje na krivulji tzv. *sampling* distribucije(distribucija vrijednosti parametra u odabranim uzorcima populacije određene veličine) koja slijedi Gauss-ovu(normalnu) distribuciju.

Jasno je da ukoliko parametar kojeg promatramo slijedi normalnu distribuciju da će i *sampling* distribucija također slijediti normalnu distribuciju. Međutim, *CLT* nam govori da čak i ako parametar koji promatramo nema normalnu distribuciju, *sampling* distribucija će imati normalan oblik ukoliko iz populacije uzimamo uzorce koji su dovoljno veliki( $n > 30$ ),

Bootstrap tehnika počiva na *CLT* teoremu te unutar uzorka kojeg posjedujemo uzima veliki broj dodatnih uzoraka na slučajan način s veličinom  $n > 30$  čije će vrijednosti parametara prema *CLT* teoremu slijediti normalnu distribuciju pa ćemo stoga moći odrediti interval povjerenja.

Način na koji ćemo odrediti interval povjerenja jest da sortiramo dobivene vrijednosti parametra u dobivenim poduzorcima i uzmemos granice intervala u ovisnosti o razini signifikantnosti.

U našem slučaju koristimo  $\alpha = 10\%$  koja je raspoređena simetrično na obje strane distribucije. Granice intervala povjerenja ćemo odrediti na sljedeći način:

- Donja granica: vrijednost za koju vrijedi da 5% uzoraka ima **manju** vrijednost parametra od zadane vrijednosti. Budući da je niz sortiran uzlazno, indeks zadane vrijednosti ćemo izračunati kao  $0.05 * \text{broj\_uzoraka}$
- Gornja granica: vrijednost za koju vrijedi da 5% uzoraka ima **veću** vrijednost parametra od zadane vrijednosti. Budući da je niz sortiran uzlazno, indeks zadane vrijednosti ćemo izračunati kao  $0.95 * \text{broj\_uzoraka}$

Detaljnije o *bootstrap* tehnici na [linku](#).

**bootstrap()** metoda kao argument prima:

- `func` : metoda koja izračunava vrijednost parametra za kojeg želimo izračunati interval povjerenja
- `a, b` : parametri potrebni za izračun vrijednosti za koju računamo interval povjerenja, proslijeduju se metodi definiranoj s parametrom `func`
- `samples` : broj poduzoraka koje ćemo kreirati *bootstrap* metodom

Metoda vraća **Python tuple** koji se sastoji od vrijednosti parametra izračunate za trenutni uzorak(tzv. *point estimate*) te donju i gornju granicu intervala povjerenja.

```
def bootstrap(a, b, func, samples=10000):
    a = np.array(a)
    b = np.array(b)
    # Niz za pohranu vrijednosti parametra svakog poduzorka za kojeg računamo interval
    bootstraps = []
    # Kreiraj onoliko uzoraka koliko je određeno samples parametrom
    for _ in range(samples):
        # Odaberis članove novih poduzoraka na slučajan način
        ind = np.random.choice(len(a), len(a))
        bootstraps.append(func(a[ind], b[ind]))
    # Sortiraj uzlazno
    bootstraps = sorted(bootstraps)

    return func(a, b), bootstraps[round(0.05 * len(bootstraps))], bootstraps[round(0.95 * len(bootstraps))]
```

## ❖ dice\_similarity\_coefficient()

Metoda služi za izračun **Sørensen–Dice koeficijenta** koji predstavlja važno mjerilo performansi našeg modela te se često koristi kao mjerilo uspješnosti modela za segmentaciju slika.

Koeficijent predstavlja mjeru sličnosti između 2 uzorka. U našem slučaju promatramo sličnost između segmentacije slike koja je rezultat našeg modela i ispravne segmentacije slike.

Ukoliko su izlazi našeg modela slični ispravnim segmentacijama slike tada naš model posjeduje dobre performanse koje će biti iskazane vrijednošću zadanog koeficijenta koja će rasti i težiti prema 1.

Koeficijent poprima vrijednosti iz intervala [0 - 1]

Formula koeficijenta uzima u obzir presjek i uniju dva uzorka:

$$DSC = \frac{2|A \cap B|}{|A| + |B|}$$

U našem slučaju želimo promatrati točnost s kojom je naš model segmentirao lijevu komoru/klijetku. Ta usporedba je moguća kod *frame*-ova za koje imamo pohranjene oznake područja lijeve komore/klijetke u *VolumeTracings.csv* datoteci. Stoga zaključujemo da se naš model trenira samo nad ovim *frame*-ovima.

Ispravni podaci segmentacije su pohranjeni unutar *target* liste svakog videa *dataset*-a u vidu *LargeTrace* i *SmallTrace* atributa koji predstavljaju bitmape. R,G,B vrijednosti za zadane *frame*-ove se također nalaze unutar *target* liste u vidu *LargeFrame* i *SmallFrame* atributa.

Zaključujemo da će naš model za ulaz primati *Large/SmallFrame* podatke te će nad njima raditi segmentaciju lijeve komore/klijetke što će ujedno biti i njegov izlaz. Rezultat našeg modela ćemo usporediti s *Large/SmallTrace* podacima i odrediti u kolikoj je mjeri naš model ispravno segmentirao lijevu komoru/klijetku.

Presjek uzoraka će u našem slučaju biti broj preklapanja piksela izlaza našeg modela i *Large/SmallTrace* na mjestima gdje se na slici nalazi lijeva komora/klijetka(vrijednost = 1) označen s *large/small\_inter* varijablama. Unija će predstavljati ukupan broj piksela postavljenih na vrijednost 1 u izlazu našeg modela i u *Large/SmallTrace* podacima što je označeno s *large/small\_union* varijablama.

Buući da se koeficijent računa na kraju svake epohe metoda će primati prethodno navedene varijable unije i presjeka u formatu liste koja će sadržavati vrijednosti unije i presjeka za svaki frame korišten u epohi.

```
def dice_similarity_coefficient(inter, union):
    return 2 * sum(inter) / (sum(union) + sum(inter))
```

▼ echo nv

•••••.py

Budući da želimo odvojiti način učitavanja i rukovanja s podacima od naše programske logike vezane uz segmentaciju slike te želimo poboljšati modularnost, čitljivost i olakšati održavanje koda, kreiran je **echo.py** modul koji definira procedure koje će biti zajedničke svakoj instanci iz *dataset-a* za treniranje, testiranje i validaciju.

Zbog kompleksnosti domene računalnog vida, pisanje programskog koda za rukovanje s podacima "od nule" bilo bi iznimno zahtjevno i dugotrajno.

Međutim, popularnost područja umjetne inteligencije i strojnog učenja u računalnom svijetu posljednih nekoliko desetljeća uz popratni eksponencijalni rast performansi hardeverskih tehnologija za posljedicu je imala razvoj različitih biblioteka i programskih okvira koji su znatno ubrzali i olakšali razvoj projekata u zadanim domenama.

U prethodno navedene programske okvire upada **PyTorch** koji je korišten u ovom projektu zajedno sa **TorchVision** bibliotekom integriranom unutar njega i specijaliziranom za područje računalnog vida.

Kao i svaki programski okvir, **PyTorch** nam pruža različite funkcionalnosti kroz svoje sučelje s ciljem ponovnog korištenja logike koja je zajednička većini projekata u domeni strojnog učenja, ali uz paralelno smanjenje fleksibilnosti budući da moramo slijediti njegova "pravila".

Upravo je takav princip korišten kod definiranja *dataloader-a* za učitavanja *dataset-a* i operacija nad samiminstancama *dataset-a*.

## ▼ PyTorch Dataloader

Osim definiranja načina učitavanja podataka za potrebe našeg modela, *Dataloader* služi kao dodatna razina apstrakcije unutar **PyTorch** okvira koja nam omogućava jednostavno iteriranje kroz učitane podatke.

Svi parametri PyTorch Dataloader konstruktora navedeni su u [dokumentaciji](#), a ovdje ćemo navesti one koji su korišteni u ovom projektu:

- **dataset** : najvažniji argument koji odgovara prethodno kreiranoj *dataset* instanci. Više detalja u **PyTorch Dataset** poglavljju.
- **batch\_size** : jedan od najvažnijih hiperparametara modela koji može drastično utjecati na njegove performanse. Definira broj instanci *dataset-a* koje se obrađuju u jednoj iteraciji. U našem slučaju ovaj broj je 20 ili 10 ovisno o situaciji.
- **num\_workers** : broj podprocesa koji će biti korišteni prilikom učitavanja podataka. Ovaj parametar omogućuje paralelizaciju učitavanja podataka. Detaljnije na [linku](#).
- **shuffle** : definira hoće li se u svakoj epohi uzimati isti podaci po redu kako su navedeni(*false*) ili će se u svakoj epohi uzimati drugčiji podaci na slučajan način(*true*). Ovaj parametar je dosta bitan ukoliko želimo izbjeći **overfittinga** modela te da pri tome

postavljamo na *true*.

- *pin\_memory* : definira hoće li učitani podaci biti kopirani u posebni *pinned memory* dio koji služi za prijenos podataka između CPU *host*-a i GPU *device*-a. Ovaj parametar može značajno utjecati na brzinu prijenosa podataka između *host*-a i *device*-a, ali naravno zahtjeva dodatne resurse kod *host*-a. Više informacija o ovom mehanizmu: [članak1](#), [članak2](#)
- *drop\_last* : definira što će se dogoditi s posljednjim *batch*-em ukoliko ukupan broj instanci *dataset*-a nije djeljiv s *batch\_size* parametrom. *True* će odbaciti zadnji *batch* iz kalkulacija, dok će *false* uzeti zadnji *batch* koji će naravno biti manje veličine od *batch\_size* parametra
- *collate\_fn* : *custom* funkcija koja spaja instance unutar *batch*-a sa specificiranom veličinom u jedan tenzor. Ukoliko ne specificiramo vlastitu metodu *Dataloader* će po *defaultu* spojiti instance *dataset*-a u formatu niza. *\_video\_collate\_fn()* je primjer *custom* metode koju koristimo prilikom testiranja te je objašnjena u nastavku.

Instanca *Dataloader* je iterator koji prolazi kroz sve podatke te u svakoj iteraciji vraća broj instanci *dataset*-a specificiran u *batch\_size* parametru unutar jednog tenzora.

#### ▼ *\_video\_collate\_fn()*

Uloga ove funkcije jest spajanje videa koji se nalaze u trenutnom *batch*-u iteracije *dataloader*a.

Za svaku instancu *dataset*-a dohvaća se tenzor koji predstavlja video te lista atributa koje smo izabrali prilikom kreiranja *dataset*-a što je specificirano unutar *\_\_getitem\_\_* metode. Video se dohvaca u formatu [channels, frame, height, width].

Međutim, spajanje videa se odnosi na spajanje *frame*-ova svih videa unutar *batch*-a u jedinstveni video po dimenziji *frame*-ova. Stoga će prva dimenzija povratnog tipa *\_video\_collate\_fn* biti dimenzija *frame*-ova za razliku od početne dimenzije R,G,B kanala.

Rezultantni tenzor dobit ćemo tako što ćemo prvo spojiti videe po dimenziji *frame*-a koristeći **NumPy** *concatenate* metode te nakon spajanja transponirati tenzor na način da dimenziju *frame*-ova postavimo kao prvu os, a dimenziju kanala kao drugu os. Vidimo da zadano transponiranje zapravo odgovara zamjeni osi pa ćemo ga implementirati korištenjem **NumPy** *swapaxes* metode.

Kako bi kasnije prilikom testiranja mogli iz spojenog videa razlikovati dijelove koji se odnose na pojedinačne videe iz *batch*-a uz proslijedivanje *video* i *target* objekata proslijedujemo i listu koja sadrži veličinu u *frame*-ovima za svaki video unutar *batch*-a po redoslijedu unutar *batch*-a. Duljine ćemo dobiti pozivom **Python** *map* metode koja kreira listu članova pri čemu je svaki član rezultat poziva metode koja se proslijedi kao prvi argument te se poziva za svaki član niza koji odgovara drugom parametru metode i koji se proslijedi kao argument *metoda*.

metode.

Također, potrebno je koristiti operator transponiranja \* kako bi pretvorili video i target tenzore u dimenzije koje su nam pogodne prilikom iteriranja batch-a.

Metoda kao jedini argument prima listu čiji je broj članova jednak broju članova instanci dataset-a unutar batch-a pri čemu su za svaku instancu definirane video i target varijable koje vraća **getitem** metoda:

- `x[i][0]` -> video varijabla i-tog videa unutar batch-a
- `x[i][1]` -> target varijabla i-tog videa unutar batch-a

Transponiranjem parametra x dobivamo tenzor od 2 retka pri čemu prvi redak predstavlja niz video varijable svih videa unutar batch-a, a drugi redak predstavlja niz target varijabli svih videa unutar batch-a.

- `video[i]` -> video varijabla i-tog videa u batch-u
- `target[i]` -> lista targeta pohranjenih za i-ti video u batch-u

Prilikom iteracije kroz trenutni batch dataloader-a jednostavniji pristup target-ima bi bio kada bi za svaki target\_type imali zasebne nizove čiji bi redoslijed članova odgovarao redoslijedu videa unutar batch-a. To postižemo jednostavnim transponiranjem target tenzora.

```
def _video_collate_fn(x):
    # Transponiraj parametar x
    video, target = zip(*x)

    # video je lista čija je duljina = batch_size, svaki član liste je video varijabla

    # target je lista čija je duljina = batch_size, svaki član liste je ponovno lista

    # Važno! -> rezolucija svih videa je ista kao i target_type-ovi

    # Niz koji sadrži broj frame-ova svakog videa
    i = list(map(lambda t: t.shape[1], video))

    # Spoji videe unutar batch-a po dimenziji frame-ova
    # Format prije: [channels, frame, height, width]
    video = torch.as_tensor(np.swapaxes(np.concatenate(video, 1), 0, 1))
    # Format nakon: [frame, channels, height, width]

    # Transponiraj target radi lakšeg rukovanja i iteriranja
    target = zip(*target)

    return video, target, i
```

## PyTorch Dataset

PyTorch podržava 2 različita tipa/formata *dataset*-ova koji se razlikuju po načinu pristupanja:

1. **map-style datasets**
2. **iterable-style datasets**

Za potrebe ovog projekta korišten je **map-style dataset** koji implementira `__getitem__()` i `__len__()` metode koje definiraju protokol pristupanja instancama sličan *key-value* pristupima unutar objekata ili *hashmap*-a.

Budući da je srž projekta segmentacija, a kako bi imali *dataset* koji je kompatibilan s **Torchvision** metodama i modelom za segmentaciju, potrebno je slijediti pravila programskog okvira koja nalažu da naš *dataset* moramo definirati kroz sučelje **torchvision.datasets.VisionDataset** klase.

Ukoliko želimo koristiti vlastiti *dataset*(a ne neki javno dostupni i ugrađen u programski okvir) tada se procedura sastoji od nasljeđivanja **torchvision.datasets.VisionDataset** klase i definiranja vlastitog konstruktora(`__init__` metoda) te vlastitih `__getitem__()` i `__len__()` metoda.

Upravo je to sadržaj **echo.py** datoteke čije ćemo dijelove objasniti u nastavku.

## ▼ `__init__()`

Sukladno objektno-orientiranoj paradigmi, funkcija konstruktora se automatski poziva prilikom kreiranja instance zadane klase(u našem slučaju je to instanca *dataset*-a).

Prilikom kreiranja instance *dataset*-a u konstruktoru postavljamo određene parametre koji se tiču cijelokupnog *dataset*-a. Parametri se proslijeđuju kao argumenti konstruktora prilikom kreiranja instance te su navedeni u nastavku:

- `root` : Root direktorij u kojem se nalazi *dataset* (*default echonet.config.DATA\_DIR*)
- `split` : Definira filter koji određuje koje instance *dataset*-a uzimamo u obzir ovisno o situaciji. To je ujedno i naziv stupca/labele u `FileList.csv` tablici koji omogućuje zadano filtriranje. Moguće vrijednosti su:
  - `train` (koriste se prilikom treniranja modela)
  - `val` (koriste se prilikom validacije modela)
  - `test` (koriste se prilikom testiranja modela)
  - `all` (uzmi sve podatke neovisno o labeli)
  - `external_test` (uzmi eksterne podatke koji se nalaze na lokaciji specificiranoj u `external_test_location`)
- `target_type` : parametar koji definira koje atribute od svih dostupnih ćemo uzeti u obzir za svaku instancu *dataset*-a. Svi mogući atributi su:
  - `Filename` : ime video datoteke

- EF(ejection-fraction) : postotak volumena u odnosu na EDV koji srce potisne prilikom kontrakcije
  - EDV(end-diastolic volume) : volumen srca u trenutku dijastole, najveći volumen prilikom ciklusa
  - ESV(end-systolic volume) : volumen srca u trenutku sistole, najmanji volumen prilikom ciklusa
  - LargeIndex : indeks *frame*-a videa u kojem se događa EDV
  - SmallIndex : indeks *frame*-a videa u kojem se događa ESV
  - LargeFrame : matrica *frame*-a videa u kojem se događa EDV s normaliziranim vrijednostima u intervalu [0-1]
  - SmallFrame : matrica *frame*-a videa u kojem se događa ESV s normaliziranim vrijednostima u intervalu [0-1]
  - LargeTrace : segmentacijska matrica LargeFrame -a u kojoj su pikseli koji su unutar lijeve komore/klijetke u EDV trenutku označeni s 1, a oni koji su izvan s 0
  - SmallTrace : segmentacijska matrica SmallFrame -a u kojoj su pikseli koji su unutar lijeve komore/klijetke u ESV trenutku označeni s 1, a oni koji su izvan s 0
- mean : aritmetička sredina za sve R,G,B kanale zajedno ili za svaki kanal odvojeno. Bitna kod normalizacije piksela videa.
  - std : standardna devijacija za sve R,G,B kanale zajedno ili za svaki kanal odvojeno. Bitna kod normalizacije piksela videa.
  - length : ukupan točan broj *frame*-ova koje ćemo izvući iz videa. Ako je None uzimamo koliko je dopušteno s max\_length parametrom.
  - period : definira *frame*-ove koje ćemo ekstrahirati iz videa. Npr, ukoliko je period=2 ekstrahirat će se *frame*-ovi 1, 3, 5, ...
  - max\_length : definira maksimalni broj *frame*-ova koje ćemo uzeti iz videa. Ako je None nema ograničenja.
  - clips : broj mini-videa koje ćemo uzorkovati unutar glavnog videa koji poštuju length, max\_length i period vrijednosti. Default 1.
  - pad : broj piksela koje je potrebno nadodati sa svake strane *frame*-a prilikom proširenja rezolucije. U tom slučaju u sredini će biti početni *frame*-ovi dok će rubovi biti obojani "srednjom" bojom *frame*-a(normalizirana vrijednost 0).
  - noise : postotak piksela kod kojih ćemo dodati/uzrokovati šum radi poboljšanja robustnosti modela i smanjenja potencijalnog overfittinga.
  - target\_transform : metoda koja prima sliku i labele te ih transformira na način zadan u metodi.
  - external\_test\_location : lokacija eksternih podatka za testiranje ukoliko je split postavljen na external\_test

Također, sukladno objektno-orientiranoj paradigmi prilikom nasljeđivanja roditeljske klase potrebno je u klasi djeteta pozvati konstruktor roditeljske klase(u našem slučaju

**torchvision.datasets.VisionDataset**). Klasi se proslijeđuju root i target\_transform parametri sukladno specifikaciji **PyTorch-a**.

Pristup instanci objekta klase za kojeg se poziva konstruktor u **Python-u** je omogućen kroz self varijablu definiranu od strane **Python** okruženja i postavljene kao prvi parametar konstruktora.

Inicijalizacija parametara koji se tiču cijelog dataset-a zajedno s predefiniranim vrijednostima parametara konstruktora i pozivom roditeljske klase prikazan je u sljedećem isječku.

```
def __init__(self, root=None,
             split="train", target_type="EF",
             mean=0., std=1.,
             length=16, period=2,
             max_length=250,
             clips=1,
             pad=None,
             noise=None,
             target_transform=None,
             external_test_location=None):
    if root is None:
        root = echenet.config.DATA_DIR

    super().__init__(root, target_transform=target_transform)

    self.split = split.upper()
    if not isinstance(target_type, list):
        target_type = [target_type]
    self.target_type = target_type
    self.mean = mean
    self.std = std
    self.length = length
    self.max_length = max_length
    self.period = period
    self.clips = clips
    self.pad = pad
    self.noise = noise
    self.target_transform = target_transform
    self.external_test_location = external_test_location
```

Nakon inicijalizacije svih parametara prvi korak je učitavanje podataka vezanih za instance dataset-a koje ćemo koristiti. Podaci su pohranjeni u 2 osnovne datoteke:

- `FileList.csv`
- `VolumeTracings.csv`

Prvo ćemo dohvati podatke iz `FileList.csv` datoteke korištenjem `pandas.read_csv()` metode koja nam omogućuje manipulaciju sa .csv datotekama. `header` atribut će se postaviti na listu imena stupaca tablice, dok će `value` atribut biti niz koji će kao članove sadržavati vrijednosti redaka tablice za svaki video također u obliku niza.

Također, moguće je specificiranjem imena stupca dobiti sve vrijednosti tog stupca korištenjem `x[ime_stupca]` sintakse.

header vrijednost ćemo spremiti u `self.header` varijablu, value u `self.outcome` te ćemo u `self.fnames` pohraniti imena svih datoteka koje sadrže videa.

```
self.fnames, self.outcome = [], []

if self.split == "EXTERNAL_TEST":
    self.fnames = sorted(os.listdir(self.external_test_location))
else:
    # Učitaj "FileList.csv" datoteku
    with open(os.path.join(self.root, "FileList.csv")) as f:
        data = pandas.read_csv(f)
    # Osiguraj da su sve vrijednosti "Split" stupca napisane velikim slovima
    data["Split"].map(lambda x: x.upper())

    if self.split != "ALL":
        # Uzmi samo one datoteke/videa koji imaju vrijednost "Split" atributa jednaku
        data = data[data["Split"] == self.split]
        # U suprotnome uzmi sve datoteke/videa

    # Imena stupaca tablice
    self.header = data.columns.tolist()
    # Imena svih datoteka/videa tablice
    self.fnames = data["FileName"].tolist()
    # Ukoliko ime nema ekstenziju po defaultu postavi da je riječ o avi formatu
    self.fnames = [fn + ".avi" for fn in self.fnames if os.path.splitext(fn)[1] == ""]
    # Tenzor koji sadržava vrijednosti atributa tablice svih redaka
    self.outcome = data.values.tolist()

    # Broj redaka tablice(videa) mora biti jednak broju dostupnih videoa odnosno svi vi
    # Kako provjeriti?
    # Duljina self.fnames mora biti jednaka broju datoteka videoa u direktoriju
    # Python -> razlika set()-ova jednaka je razlici broja elemenata set()-ova
    missing = set(self.fnames) - set(os.listdir(os.path.join(self.root, "Videos")))
    if len(missing) != 0:
        print("{} videos could not be found in {}".format(len(missing), os.path.join(
            for f in sorted(missing):
                print("\t", f)
        raise FileNotFoundError(os.path.join(self.root, "Videos", sorted(missing)[0])))
```

Time smo učitali sve podatke vezane uz `Filelist.csv` datoteku.

Značenja atributa tablice iz `VolumeTracings.csv` datoteke nisu intuitivna za razliku od onih iz `Filelist.csv` datoteke te su usko vezani uz problematiku koju pokušavamo rješiti u našem slučaju.

FileName	X1	Y1	X2	Y2	Frame
0X100009310A3BD7	51.26041667	15.34895833	64.93229167	69.125	46
0X100009310A3BD7	50.03761083	17.16784126	53.36722189	16.32132997	46
0X100009310A3BD7	49.15727221	20.40762020	57.00051825	18.20072116	46

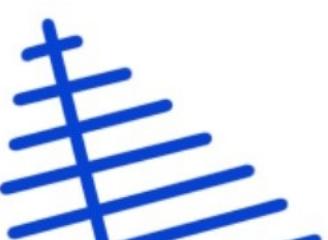
0X100009310A3BD7	48.5381733	23.58105455	59.99733911	20.66770731	46
0X100009310A3BD7	47.91896839	26.7544797	62.90412937	22.94469301	46
0X100009310A3BD7	47.9621045	29.75951307	65.81091963	25.22167871	46
0X100009310A3BD7	48.16791529	32.72318847	68.24704281	27.61832554	46
0X100009310A3BD7	48.37372608	35.68686387	70.38531105	30.0906982	46
0X100009310A3BD7	48.57953687	38.65053927	72.5235793	32.56307085	46
0X100009310A3BD7	49.01403934	41.55607271	74.15164369	35.16515635	46
0X100009310A3BD7	49.83099892	44.3643713	75.71299787	37.78420208	46
0X100009310A3BD7	50.64795851	47.17266989	77.27435205	40.4032478	46
0X100009310A3BD7	51.4649181	49.98096847	78.64469427	43.07085589	46
0X100009310A3BD7	52.54732826	52.72177963	79.48537463	45.87312377	46
0X100009310A3BD7	53.7042693	55.44364225	80.326055	48.67539165	46
0X100009310A3BD7	54.36563	58.29149988	80.609375	51.61936132	46
0X100009310A3BD7	51.15135239	62.12468928	80.609375	54.63536149	46
0X100009310A3BD7	49.33671909	65.6020369	80.4312144	57.69665674	46
0X100009310A3BD7	50.3482871	68.36085877	80.06545076	60.80564767	46
0X100009310A3BD7	57.51297555	69.55532799	79.69968713	63.9146386	46
0X100009310A3BD7	71.93835296	68.90385933	79.33392349	67.02362953	46
0X100009310A3BD7	56	19.54166667	61.65104167	62.74479167	61
0X100009310A3BD7	55.61529756	20.69577397	57.09330327	20.50244833	61
0X100009310A3BD7	54.99178009	22.98490627	59.27990982	22.42401166	61

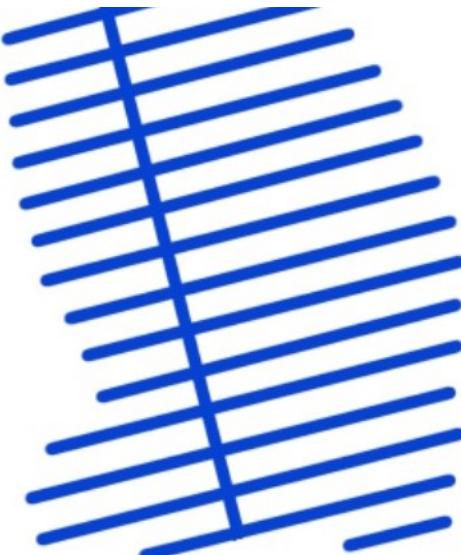
Prisjetimo se, naš cilj je segmentirati lijevu komoru/klijetku srca na videozapisu. Kako bi naš model mogao "naučiti" razlikovati lijevu klijetku od ostatka srca mi ga moramo istrenirati na način da mu na velikom broju slika označimo gdje se nalazi lijeva klijetka. Upravo je to uloga VolumeTracings.csv datoteke.

Međutim, preostaje jedno važno pitanje: Kako računalu na efektivan način predstaviti gdje se na slici/frame-u nalazi lijeva komora/klijetka?

Neki početni bruteforce odgovor bi bio da svakoj slici pridružimo bitmapu koja će s 1 označiti piksele koji predstavljaju lijevu komoru a s 0 ostale piksele. Lako je zaključiti da je ovakav pristup iznimno neefikasan kako memorijski tako i vremenski u vidu labeliranja podataka.

U ovom projektu je stoga korištena kreativna ideja u vidu predstavljanja lijeve komore/klijetke dužinama kao što je prikazano na idućoj slici:





Svaka dužina definirana je s 2 točke od kojih svaka ima vlastitu x i y koordinatu unutar koordinatnog sustava slike. Ukoliko povežemo točke dužina ravnim linijama dobit ćemo poligon koji omeđuje lijevu komoru/klijetku srca te ćemo njenu poziciju na slici moći predočiti računalu.

VolumeTracings.csv datoteka sadrži koordinate prve(X1,Y1) i druge(X2,Y2) točke dužina grupiranih po imenu videa definiranog u `FileName` stupcu i `frame`-u čiji je redni broj definiran u `Frame` stupcu.

Način označavanja lijeve komore/klijetke na slici će biti detaljnije obrađen kasnije.

Podatke iz VolumeTracings.csv datoteke ćemo spremati u `self.frames` i `self.trace` atributе koji će biti `collections.defaultdict Python tipa` koji radi na `key-value` principu.

```
self.frames = collections.defaultdict(list)
self.trace = collections.defaultdict(_defaultdict_of_lists)

def _defaultdict_of_lists():
    return collections.defaultdict(list)
```

Vidimo da će `self.frames` za `value` dio sadržavati listu, dok će `self.trace` za `value` dio sadržavati drugi `dictionary` koji će zatim za svoj `value` dio sadržavati listu. Ove razine indeksiranja bit će jasnije kada objasnimo značenje ovih atributa.

Uloga `self.frames` atributa je da za svaku datoteku/video za koji imamo podatak u VolumeTracings.csv datoteci zapišemo listu rednih brojeva `frame`-ova za koje su definirani

podaci. *Key* ce biti ime datoteke dok ce *value* biti lista indeksa *frame*-ova.

Uloga **self.trace** atributa je da za svaku datoteku/video za koji imamo podatak u *VolumeTracings.csv* datoteci za svaki njegov *frame* zapišemo listu u kojoj će svaki član biti niz od 4 člana koji predstavljaju *x1*, *y1*, *x2*, *y2* koordinate točaka dužine. Svi članovi liste za zadani *frame* predstavljaju potpuni prikaz/oznaku lijeve komore/klijetke srca na zadanom *frame*-u.

Programski kod za inicijalizaciju ovih atributa je sljedeći:

```
with open(os.path.join(self.root, "VolumeTracings.csv")) as f:
    header = f.readline().strip().split(",")
    assert header == ["FileName", "X1", "Y1", "X2", "Y2", "Frame"]
    for line in f:
        # line = redak tablice, vrijednosti retka su odvojene zarezom pa ih parsiramo pr
        filename, x1, y1, x2, y2, frame = line.strip().split(',')
        x1 = float(x1)
        y1 = float(y1)
        x2 = float(x2)
        y2 = float(y2)
        frame = int(frame)
        if frame not in self.trace[filename]:
            # Dodaj redni broj frame-a u za zadani file za kojeg imamo podatke
            self.frames[filename].append(frame)
        # Dodaj koordinate dužine za zadani file i zadani frame u formatu Python liste
        self.trace[filename][frame].append((x1, y1, x2, y2))
    for filename in self.frames:
        for frame in self.frames[filename]:
            # Pretvori iz liste u NumPy niz
            self.trace[filename][frame] = np.array(self.trace[filename][frame])

# Zadrzi podatke samo za ona videa koji imaju podatke u VolumeTracings.csv datoteci
# U slučaju da neki videi nemaju zadane tracing podatke
keep = [len(self.frames[f]) >= 2 for f in self.fnames]
self.fnames = [f for (f, k) in zip(self.fnames, keep) if k]
self.outcome = [f for (f, k) in zip(self.outcome, keep) if k]
```

Time je završena obrada uloge i same implementacija **\_\_init\_\_()** konstruktora našeg *dataset*-a.

## ▼ **\_\_getitem\_\_()**

Kao što je prethodno rečeno, u našem projektu koristimo **map-style dataset** u kojem podacima pristupa preko *key-value* mehanizma. Prilikom učitavanja svake instance *dataset*-a **PyTorch** će pozvati ovu metodu za svaku instancu te joj automatski proslijediti *index* parametar koji označava indeks(redni broj-1) videa unutar direktorija iz kojeg se učitavaju podaci.

Prvi korak u metodi je učitavanje videa specificiranog zadanim indeksom. Budući da je

redoslijeda videa u direktoriju jednako redoslijedu redaka u `filelist.csv` datoteci možemo koristiti `self.fnames` listu definiranu unutar konstruktora koja će nam za zadani `index` vratiti ime datoteke/videa koje nam je potrebno za njegovo učitavanje. Učitavanje videa ćemo raditi korištenjem `loadvideo()` utility funkcije.

Nakon poziva `loadvideo()` metode imamo tenzor koji sadrži podatke o R,G,B vrijednostima za svaki *frame* i svaki piksel na *frame*-u i to u [channels, frame, height, width] formatu.

```
def __getitem__(self, index):
    #index= INDEX videa u direktoriju odakle se dohvaćaju podaci
    # Find filename of video
    if self.split == "EXTERNAL_TEST":
        video = os.path.join(self.external_test_location, self.fnames[index])
    elif self.split == "CLINICAL_TEST":
        video = os.path.join(self.root, "ProcessedStrainStudyA4c", self.fnames[index])
    else:
        video = os.path.join(self.root, "Videos", self.fnames[index])

    # Load video into np.array
    #format [channels, frame, height, width]
    video = echonet.utils.loadvideo(video).astype(np.float32)
```

Idući korak je provjera potrebe za ubacivanjem šuma u učitane podatke ukoliko je to bilo definirano s `noise` parametrom konstruktora koji definira postotak piksela na kojima će biti kreiran/dodan šum.

Šum se sastoji u postavljanju crne boje odnosno vrijednosti 0 za R,G,B vrijednosti budući da vrijednosti boja još nisu normalizirane u ovom trenutku.

Budući da `noise` parametar predstavlja postotak piksela u videu kod kojih ćemo postaviti šum prvo moramo izračunati ukupan broj piksela videa koji se dobije prema formuli:

```
broj_pixela = broj_frameova * broj_redaka * broj_stupaca
```

Od ukupnog broja pixela u njih `self.noise * broj_pixela` će biti umetnut šum. Budući da je i ideja samog šuma u slučajnoj smetnji na slučajan način ćemo izabrati `self.noise * broj_pixela` indeksa u rasponu od 0 do `broj_pixela`.

Nakon odabira indeksa budući da je naš video objekt u formatu [channel, frame, height, width] moramo se vratiti u njegovu indeksaciju odnosno iz odabranog indeksa dobiti informaciju o kojem je *frame*-u riječ i o kojim je x,y koordinatama piksela unutar *frame*-a riječ.

```
if self.noise is not None:
    # Ukupan broj pixela u cijelom videu
    n = video.shape[1] * video.shape[2] * video.shape[3]
    # Odaberite na random način indekse piksela u koje ćemo unijeti šum
    # Biramo self.noise * n indeksa u rasponu 0- n
```

```

ind = np.random.choice(n, round(self.noise * n), replace=False)
# Vrati se u [c,f,h,w] indeksaciju
f = ind % video.shape[1]
ind //= video.shape[1]
i = ind % video.shape[2]
ind //= video.shape[2]
j = ind
# Postavi R,G,B vrijednosti na 0 -> crna boja = šum
video[:, f, i, j] = 0

```

Nakon optionalnog ubacivanja šumova slijedi obvezna normalizacija R,G,B vrijednosti na raspon [0-1] i to korištenjem formule za jediničnu normalnu varijablu z:

$$z = \frac{(x - \mu)}{\sigma}$$

Uočavamo da su za normalizaciju potrebne prethodno izračunate aritmetička sredina i standardna devijacija izračunate posebno za svaki kanal i proslijeđene u konstruktor. Kako bi mogli dijeliti video tenzor formata [channel, frame, height, width] po svakom kanalu s odgovarajućom aritmetičkom sredinom i standardnom devijacijom za zadani kanal potrebno je pretvoriti aritmetičku sredinu i standardnu devijaciju u tenzore dimenzija [3, 1, 1, 1]

```

if isinstance(self.mean, (float, int)):
    video -= self.mean
else:
    video -= self.mean.reshape(3, 1, 1, 1)

if isinstance(self.std, (float, int)):
    video /= self.std
else:
    video /= self.std.reshape(3, 1, 1, 1)

```

Sljedeći korak je osiguravanje kriterija postavljenih s `length`, `max_length` i `period` parametrima konstruktora.

Prisjetimo se da `length` definira točan broj `frame`-ova koje će sadržavati video(ukoliko je specificiran, u suprotnome uzimamo sve `frame`-ove koje možemo) dok `max_length` definira maksimalan broj `frame`-ova koje video može sadržavati.

Broj `frame`-ova koje ćemo uzeti u razmatranje ovisi o `period` parametru koji definira period uzorkovanja `frame`-ova videa. Npr. ukoliko naš video ima 20 `frame`-ova, a uzorkujemo svaki drugi `frame`, tada će resultantni video imati  $20 / 2 = 10$  `frame`-ova.

Također, potrebno je osigurati da veličina krajnjeg rezultatnog videa bude manja od `max_length` vrijednosti, ali u isto vrijeme i barem jednaka `length` vrijednosti ukoliko je ona specificirana.

Ukoliko je veća od `max_length` uzimamo samo prvih `max_length` `frame`-ova.

Ukoliko je manja od `length` padd-amo video na njegovom kraju na način da `frame`-ove koji

nedostaju dodamo na kraj videa te njihove R,G,B vrijednosti postavimo na 0.

Napomena: za razliku od umetanja šuma u ovom trenutku R,G,B vrijednosti su normalizirane pa vrijednost 0 predstavlja vrijednost aritmetičke sredine izvornih R,G,B boja.

```
c, f, h, w = video.shape
if self.length is None:
    # length označava broj frame-ova uzorkovanog videa
    length = f // self.period
else:
    # Rezultantni video mora imati točno length frame-ova
    length = self.length

if self.max_length is not None:
    # Uzmi prvih max_length frame-ova ako je length > max_length
    length = min(length, self.max_length)

#Broj frameova rezultantog videa manji od broja frameova specificiranog u length(za slu
if f < length * self.period:
    # Paddamo video s frame-ovima pri kraju i to za onoliko frame-ova koliko nam nedos
    video = np.concatenate((video, np.zeros((c, length * self.period - f, h, w)), video
c, f, h, w = video.shape # pylint: disable=E0633
```

Sljedeći dio koda je vezan uz uzorkovanje i dijeljenje videa u manje dijelove tzv. *clipove*. Ova značajka može biti korisna za povećanje broja instanci na način da iz jednog videa dobijemo veći broj manjih videa i time umjetno povećamo broj videa na način da unutar jednog video objekta imamo više zasebnih clipova. Ovo može biti korisno prilikom testiranja modela jer tada izvodimo predikcije na odabranim tipovima podataka, a ovim postupkom povećavamo veličinu testnog seta.

Ukoliko je parametar postavljen na ključnu riječ `all` tada uzimamo sve moguće *clip-ove* unutar videa. Npr. svi mogući clipovi duljine 5 frame-ova unutar videa duljine 10 frame-ova bili [1,2,3,4,5], [2,3,4,5,6], ... Zaključujemo da se samo pomičemo za jedan *frame* udesno. Krajnja granica do koje se možemo pomicati je onaj *frame* nakon kojeg postoji još `length` *frame-ova* do kraja videa odnosno:

```
zadnji_indeks = broj_frameova - (length - 1) * period_uzorkovanja
```

što je upravo i definirano u kodu.

Ukoliko je `self.clips` postavljen na broj tada iz videa želimo izvući `self.clips` *clip-ova* duljine `length`. Logika uvjeta za zadnji indeks je ista samo što za razliku od `all` slučaja ne uzimamo sve indekse u rasponu od `[0 - zadnji_indeks]` već na slučajan način u tom rasponu biramo `self.clips` početnih indeksa. Ukoliko je `self.clips` postavljen na 1 te uzorkujemo cijeli video(`length = None`) tada će rezultat biti cijeli video uzorkovan specificiranom frekvencijom

uzorkovanja.

Odabrane indekse početaka *clip*-ova spremamo u *start* niz.

```
if self.clips == "all":  
    start = np.arange(f - (length - 1) * self.period)  
else:  
    start = np.random.choice(f - (length - 1) * self.period, self.clips)
```

Prilikom definiranja argumenata konstruktora spomenuli smo *target\_type* parametar koji definira attribute koje ćemo pridružiti svakoj instanci *dataset*-a/videu. Mogući atributi su prethodno navedeni prilikom definiranja parametara konstruktora, a u nastavku su prikazani načini njihove inicijalizacije:

- **Filename**: možemo ga lako dohvatiti iz **self.fnames** preko *indexa* koji se proslijeđuje **\_\_getitem\_\_** metodi
- **LargeIndex**: za svaki video imamo pohranjene indekse svih *frame*-ova koji imaju podatke u *VolumeTracings.csv* datoteci u **self.frames**. Budući da je riječ o dijastoli, podaci za dijastolu će u tablici uvijek biti posljednje navedeni pa uzimamo zadnji član **self.frames** niza za video određen *indexom*.
- **SmallIndex**: ista logika kao za **LargeIndex** samo što su u ovom slučaju podaci za sistolu navedeni odmah na početku pa uzimamo zadnji član **self.frames** niza za video određen *indexom*.
- **LargeFrame**: podaci za *frame* na **LargeIndex** poziciji. Budući da smo dohvatili video u [channels, frame, height, width] formatu za dohvat podataka potrebno je samo specificirati indeks *frame*-a, a on je identičan kao **LargeIndex**
- **SmallFrame**: podaci za *frame* na **SmallIndex** poziciji. Budući da smo dohvatili video u [channels, frame, height, width] formatu za dohvat podataka potrebno je samo specificirati indeks *frame*-a, a on je identičan **SmallIndex**
- **LargeTrace** i **SmallTrace**: želimo dohvatiti podatke iz *VolumeTracings.csv* datoteke za *frame* definiran s **LargeIndex/SmallIndex**. Prethodno smo u konstruktoru pohranili *VolumeTracings.csv* podatke u **self.trace** *dictionary* grupirane po imenu videa i indeksu *frame*-a pri čemu su podaci pohranjeni u obliku liste 4-članih nizova (x1, y1, x2, y2). U našem slučaju imamo definirano ime videa kroz *index* parametar te znamo indeks *frame*-a preko **LargeIndex/SmallIndex** varijable pa stoga možemo na jednostavan način dohvatiti podatke o dužinama za zadani *frame*.

Ovdje ćemo se još malo zadržati na **LargeTrace** i **SmallTrace** *target*-ima. Naime, predstavljanje lijeve komore/klijetke korištenjem dužina je iznimno domišljato i efektivno rješenje prilikom definiranja *dataset*-a. Međutim, računalo će na koncu raditi s pikselima slike te ga moramo istrenirati kako bi na zadanom *frame*-u uočilo i segmentiralo lijevu komoru/klijetku srca.

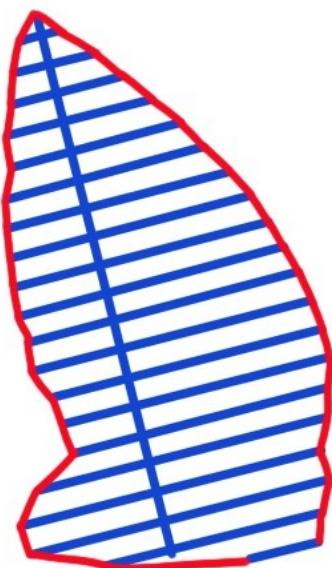
Osnova svakog treniranja modela jest postavljanie ulaza u neuralnu mrežu te usporedba

rezultata/izlaza mreže s očekivanim rezultatima. Na osnovu izlazne greške model ažurira svoje težine preko tzv. *backpropagation* postupka s ciljem smanjivanja greške uz uvijek prisutni potencijalni rizik od *overfitt-a*.

Izlaz segmentacijskog modela u našem slučaju će biti bitovna matrica svakog *frame-a* u kojoj će vrijednost 1 imati pikseli za koje model smatra da predstavljaju lijevu komoru/klijetku, a ostali pikseli će biti 0.

Kako bi mogli na efektivan način usporediti izlaze modela sa željenim izlazima koji su opisani preko modela dužina u *VolumeTracings.csv* datoteci potrebno je zadani model dužina također pretočiti u prikaz bitovne matrice.

Spajanjem točaka dužina ravnim linijama dobit ćemo poligon koji predstavlja lijevu komoru/klijetku. Zadani postupak prikazan je na sljedećoj slici crvenom linijom:



Niz 4-članih nizova formata [x1, y1, x2, y2] pretvorimo u 4 zasebna niza od kojih će svi biti jednake dužine te će zasebno predstavljati x1, y1, x2 i y2 koordinate.

Bitan faktor u našem postupku povezivanja točaka dužina jest svojstvo *dataset-a* koji je definiran na način da su točke u *VolumeTracings.csv* datoteci sortirane uzlazno po Y1 parametru.

Budući da je ishodište koordinatnog sustava slike u **gornjem lijevom kutu** zaključujemo da su točke (x1,y1) točke koje definiraju lijevi dio komore dok su točke (x2,y2) točke koje definiraju desnu stranu komore pri čemu vrijedi  $Y2 < Y1$  zbog čega su dužine nagnute kako je prikazano na prethodnoj slici. Točke (x1,y1) su sortirane uzlazno po Y1 parametru što znači da su navedene od vrha prema dnu slike.

Također, prve 2 točke (x1,y1) i (x2,y2) označavaju glavnu središnju liniju(vidljivu na slici) koja nam prilikom iscrtavanja poligona nije bitna pa stoga nećemo uzeti u obzir koordinate tih točaka.

Krenemo s iscrtavanjem od prve/najgornje točke (x1,y1) te povezujemo sve (x1,y1) točke s lijeve strane komore.

Nakon što smo došli do zadnje točke potrebno je nastaviti s povezivanjem (x2,y2) točaka na desnoj strani komore.

Međutim, naši nizovi X1, Y1, X2, Y2 su definirani prema redoslijedu sparivanja (x1,y1) i (x2,y2) dužina odnosno prvi članovi X2 i Y2 nizova odgovaraju točki dužine koja počinje s najgornjom točkom (x1,y1).

Nakon iscrtavanje lijeve strane komore nalazimo se na posljednjoj/najdonjoj točki. Kako bi nastavili s povezivanjem poligona iduća točka bi trebala biti najdonja točka desne strane komore koja odgovara zadnjim članovima nizova X2 i Y2 te ćemo povezivanje nastaviti sve do najgornje točke.

Stoga zaključujemo da ćemo prilikom iscrtavanja **invertirati** nizove X2 i Y2.

Za iscrtavanje poligona koristit ćemo `skimage.draw.polygon` metodu koja prima nizove X i Y koordinata točaka koje se povezuju po redoslijedu nizova te dimenzije slike/matrice u čijem koordinatnom sustavu se odvija iscrtavanje. Bitna stvar ove metode je njena povratna vrijednost koja odgovara nizovima X i Y koordinata točaka koji se nalaze **unutar** iscrtanog poligona što u našem konkretnom slučaju predstavlja područje lijeve komore/klijetke.

Bitmapu generiramo na način da prvo inicijaliziramo matricu veličine *frame-a* na vrijednosti 0 te one piksele čije je koordinate vratila `skimage.draw.polygon` metoda postavimo na 1.

Programska implementacija prethodno navedenih principa je realizirana sljedećim programskim kodom:

```
# Postavi specificirane targete
target = []
for t in self.target_type:
    key = self.fnames[index]
```

```

if t == "Filename":
    target.append(self.fnames[index])
elif t == "LargeIndex":
    # Index = -1 -> posljednji član
    target.append(np.int(self.frames[key][-1]))
elif t == "SmallIndex":
    target.append(np.int(self.frames[key][0]))
elif t == "LargeFrame":
    target.append(video[:, self.frames[key][-1], :, :])
elif t == "SmallFrame":
    target.append(video[:, self.frames[key][0], :, :])
elif t in ["LargeTrace", "SmallTrace"]:
    if t == "LargeTrace":
        t = self.trace[key][self.frames[key][-1]]
    else:
        t = self.trace[key][self.frames[key][0]]
    #Zasebni nizovi koordinata točaka
    x1, y1, x2, y2 = t[:, 0], t[:, 1], t[:, 2], t[:, 3]
    # Invertiraj x2 i y2 nizove
    x = np.concatenate((x1[1:], np.flip(x2[1:])))
    y = np.concatenate((y1[1:], np.flip(y2[1:])))
    r, c = skimage.draw.polygon(np.rint(y).astype(np.int), np.rint(x).astype(np.int))
    #Veličina maske = rezolucija videa
    mask = np.zeros((video.shape[2], video.shape[3]), np.float32)
    #Pixeli lijeve komore/klijetke = 1
    mask[r, c] = 1
    # Pohrani masku
    target.append(mask)
else:
    # Nije od prevelike važnosti u našem slučaju
    if self.split == "CLINICAL_TEST" or self.split == "EXTERNAL_TEST":
        target.append(np.float32(0))
    else:
        target.append(np.float32(self.outcome[index][self.header.index(t)]))

if target != []:
    # Ukoliko je definirana target_transform metoda u konstruktoru transformiraj preth
    target = tuple(target) if len(target) > 1 else target[0]
    if self.target_transform is not None:
        target = self.target_transform(target)

```

Posljednji dio **getitem** metode tiče se uzorkovanja videa odnosno njegovog dijeljenja u manje isječke/*clip*-ove te *padd*-anja videa ukoliko je to definirano u pozivu konstruktora. Naime, ukoliko je video duži od točno specificirane duljine *length* ili iz jednog videa želimo uzeti više isječaka/*clip*-ova tada to možemo odraditi na više načina.

Ovaj dio veže se na prethodni dio određivanja *start* pozicija *clip*-ova koje definiraju početne indekse *frame*-ova *clip*-ova.

Svaki *clip* će imati početni *frame* definiran unutar *start* niza te će biti dužine *length*. Npr, ukoliko početni video ima 100 *frame*-ova, *clip* je duljine 5 *frame*-ova, indeks početnog *frame*-a je 6, a želimo uzorkovati svako drugi *frame*(**self.period=2**) tada će se naš *clip* sastojati od

*frame*-ova s indeksima 6, 8, 10, 12, 14 unutar početnog videa.

Ukoliko je broj *clip*-ova postavljen na 1(*default*) tada ćemo kao rezultat imati jedan *clip* kao vrijednost video varijable, a u slučaju više *clip*-ova video varijabla će sadržavati *clip*-ove jedan iza drugog.

Ukoliko želimo povećati rezoluciju videa to možemo odraditi jednostavnim mehanizmom *paddinga* čije dimenzije određuje **self.pad** parametar. Mehanizam se sastoji od "uokviravanja" trenutnog *frame*-a s okvirom čije se normalizirane R,G,B vrijednosti postavljaju na 0(srednja vrijednost boje) dok se središnji pikseli slike postavljaju na vrijednosti trenutnog *frame*-a.

Na kraju od proširene slike *frame*-a uzimamo/cropp-amo dio veličine početne rezolucije *frame*-a.

```
# video = lista clipova
video = tuple(video[:, s + self.period * np.arange(length), :, :] for s in start)
if self.clips == 1:
    #Samo 1 clip napravljen -> uzmi prvi i jedini član liste
    video = video[0]
else:
    # Spoji clipove jedan iza drugog u jedinstveni niz
    video = np.stack(video)

if self.pad is not None:
    # Početna rezolucija
    c, l, h, w = video.shape
    # Proširi rezoluciju za "okvire" sa svih strana frame-a
    temp = np.zeros((c, l, h + 2 * self.pad, w + 2 * self.pad), dtype=video.dtype)
    # Područja "unutar okvira" postavi na vrijednosti od frame-ova od videa
    temp[:, :, self.pad:-self.pad, self.pad:-self.pad] = video
    i, j = np.random.randint(0, 2 * self.pad, 2)
    # Random croppaj dio veličine početne rezolucije videa
    video = temp[:, :, i:(i + h), j:(j + w)]
# Vrati video i target u obliku Python tuple tipa
return video, target
```

## ▼ **\_len\_()**

Metoda koju je potrebno definirati prilikom definiranja vlastitog dataset-a u **PyTorchu**. Vraća broj instanci dataset-a. U našem slučaju su imena svih instanci pohranjeni u **self.fnames** pa je dovoljno uzeti duljinu te liste.

```
def __len__(self):
    #broj videa=broj filenameova
    return len(self.fnames)
```

## ▼ **\_\_extra\_repr\_\_**

Definira dodatne informacije koje se dodaju na kraj **Python \_\_repr\_\_ stringa** za zadani dataset objekt.

```
def extra_repr(self) -> str:  
    """Additional information to add at end of __repr__. """  
    lines = ["Target type: {target_type}", "Split: {split}"]  
    return '\n'.join(lines).format(**self.__dict__)
```

```
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-generic-recommen
```

```
!jupyter nbconvert --to latex SegmentacijaEhokardiograma.ipynb
```