

Wojciech Szewczyk

Rafał Zabłotni

Kamień milowy 1: Określenie tematu i celu projektu, analiza wymagań

- Zbiór danych: Water Meters (wodomierze)
- Struktura katalogów:
 - Data → images
→ masks
- Format zapisu (image + mask (ta sama nazwa)): id_NR_ID_value_V1_V2

gdzie NR_ID, V1, V2 wartości zmienne

- Format plików: JPG
- Rozmiar obrazów: różny (wymagane skalowanie)
- Ilość obrazów: 1244 + tyle samo masek
- Format obrazów: RGB
- Format masek: greyscale(?)
- Odpowiednia ilość: tak (tyle samo obrazów i masek)
- Konfiguracja środowiska lokalnego: 'conda list' →

pytorch	2.5.1	py3.12_cpu_0	pytorch
pytorch-mutex	1.0	cpu	pytorch
pyyaml	6.0.2	py312h827c3e9_0	
qt-main	5.15.2	h19c9488_12	
requests	2.32.3	py312haa95532_1	
scikit-learn	1.6.1	py312h585ebfc_0	
scipy	1.15.1	py312hbb039d4_0	
setuptools	75.8.0	py312haa95532_0	
sip	6.7.12	py312h5da7b33_1	
six	1.16.0	pyhd3eb1b0_1	
sqlite	3.45.3	h2bbff1b_0	
sympy	1.13.3	py312haa95532_1	
tbb	2021.8.0	h59b6b97_0	
threadpoolctl	3.5.0	py312hfc267ef_0	
tk	8.6.14	h0416ee5_0	
torchaudio	2.5.1	py312_cpu	pytorch
torchvision	0.20.1	py312_cpu	pytorch

PyTorch version: 2.5.1

Torchvision version: 0.20.1

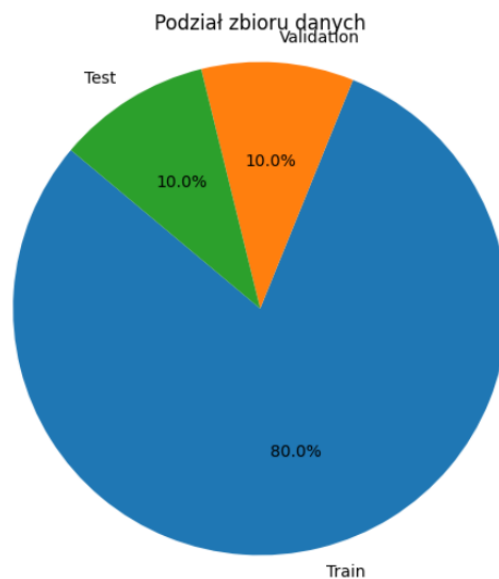
Wykrycie GPU przez PyTorch: tak

Kamień milowy 2: Zbiór danych i przygotowanie go

Wczytujemy wcześniej pobrany zestaw zdjęć i masek, a następnie dzielimy go na 3 foldery:

- Treningowy (train): Zbiór na którym będziemy uczyć nasz model
- Walidacyjny (val): Do dostrajania hiperparametrów i monitorowania, czy model odpowiednio się uczy lub się nie przeuczył (overfitting – zna wszystko na pamięć)
- Testowy (test): Dla ostatecznej oceny modelu na danych, których model wcześniej nie widział.

Dzięki ustalonemu ziarnu losowemu, zbiór będzie losowy, ale powtarzalnie, ponieważ wszystkie operacje losowe będą się zachowywać tak samo za każdym razem. Poniżej podział danych:



W celu zapewnienia poprawnego formatu danych, wszystkie zdjęcia są skalowane do rozmiaru 512x512. Następnie przekształcamy ją na tablicę NumPy w postaci float32 w zakresie [0,1], stosujemy rozciąganie kontrastu (wykorzystując 2. i 98. percentyl pikseli) oraz filtr medianowy 3x3 (po skali do uint8), a na końcu zamieniamy wynik na tensor PyTorch w formacie CHW.

```
def to_float_np(img): 1 usage 1 vvszewczyk
    arr = np.array(img).astype(np.float32) / 255.0
    return arr

def contrast_stretch(img: np.ndarray): 1 usage 1 vvszewczyk
    p2, p98 = np.percentile(img, [2, 98])
    img = (img - p2) / (p98 - p2 + 1e-6)
    return np.clip(img, a_min=0.0, a_max=1.0)

def median_blur(img: np.ndarray): 1 usage 1 vvszewczyk
    u8 = (img * 255).astype(np.uint8)
    blurred = cv2.medianBlur(u8, ksize=3)
    return blurred.astype(np.float32) / 255.0

imageTransforms = TRANS.Compose([
    TRANS.ToPILImage(),
    TRANS.Resize((512, 512)),
    TRANS.Lambda(to_float_np),
    TRANS.Lambda(contrast_stretch),
    TRANS.Lambda(median_blur),
    TRANS.Lambda(lambda arr: torch.from_numpy(arr).permute(2, 0, 1)), # HWC -> CHW
])
```

Nasz zestaw danych reprezentujemy klasą typu dataset pod nazwą WMSDataset. **PyTorch** wymaga, aby każda klasa zestawu danych dziedziczyła po Dataset. W związku z tym, nasza klasa musi implementować metody `__init__` (pobiera niezbędne informacje do wczytania zdjęć) i `__getitem__` (wczytuje każde zdjęcie). Dodatkowo dodano metodę `__len__` zwracającą ilość zdjęć.

```
class WMSDataset(Dataset): 10 usages  ▴ vvszewczyk +1
    def __init__(self, imagePaths, maskPaths, imageTransforms):  ▴ vvszewczyk
        self.imagePaths = imagePaths
        self.maskPaths = maskPaths
        self.imageTransforms = imageTransforms

    def __len__(self):  ▴ vvszewczyk
        return len(self.imagePaths)

    def __getitem__(self, i):  ▴ vvszewczyk +1
        image_path = self.imagePaths[i] #Grab the image path form the current index
        image = cv2.imread(image_path) # Load image
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        if self.imageTransforms:
            image = self.imageTransforms(image)

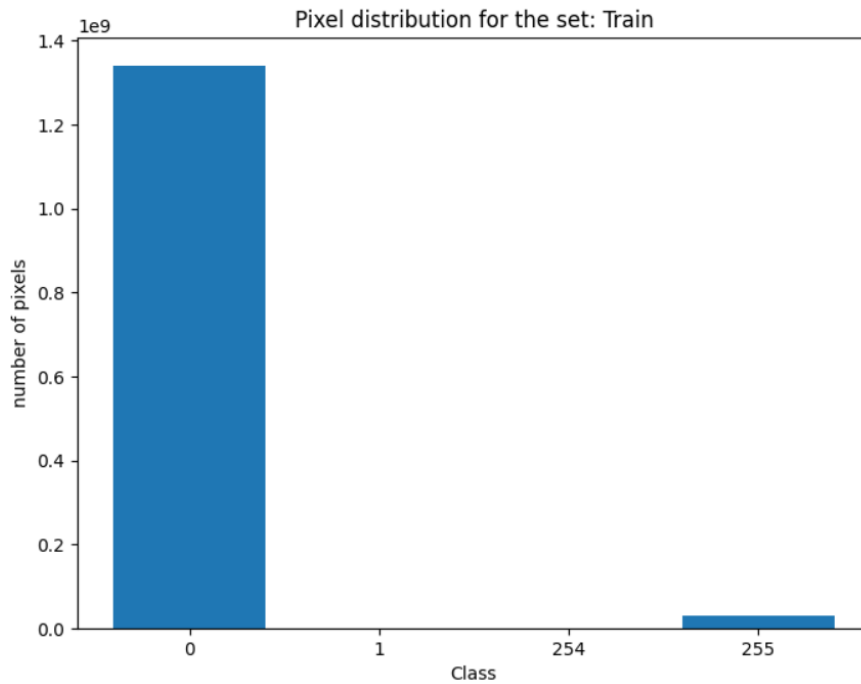
        mask = cv2.imread(self.maskPaths[i], cv2.IMREAD_GRAYSCALE) # uint8
        # Threshold to 0/1
        mask = (mask >= 127).astype(np.uint8)
        # resize from nearest neighbor
        mask = cv2.resize(mask, dsize=(512, 512), interpolation=cv2.INTER_NEAREST)
        # To tensor float32 0.0/1.0
        mask = torch.from_numpy(mask.astype(np.float32))[None, ...] # 1xHxW

        return image, mask
```

Wracając do walidacji, dodano funkcje której celem jest zliczenie wystąpień poszczególnych wartości pikseli (klas) we wszystkich maskach.

```
def count_pixel_balance(mask_paths, dataset_name): 3 usages  ▴ Rafalost *
    counts = defaultdict(int)
    for mask_path in mask_paths:
        mask = cv2.imread(mask_path, 0) # Wczytujemy maskę jako grayscale
        if mask is None:
            print(f"Warning: Could not load {mask_path}")
            continue
        unique, cnts = np.unique(mask, return_counts=True)
        for cls, count in zip(unique, cnts):
            counts[cls] += count
    print(f"\nPixel distribution for the set {dataset_name}:")
    for cls, count in sorted(counts.items()):
        print(f"Class {cls}: {count} pixels")
```

Wykres rozkładu pikseli masek grupy "Train"



Maskę ładujemy jako obraz grayscale, progowo binaryzujemy (0/1 przy progu 127),

```
mask = (mask >= 127).astype(np.uint8)
```

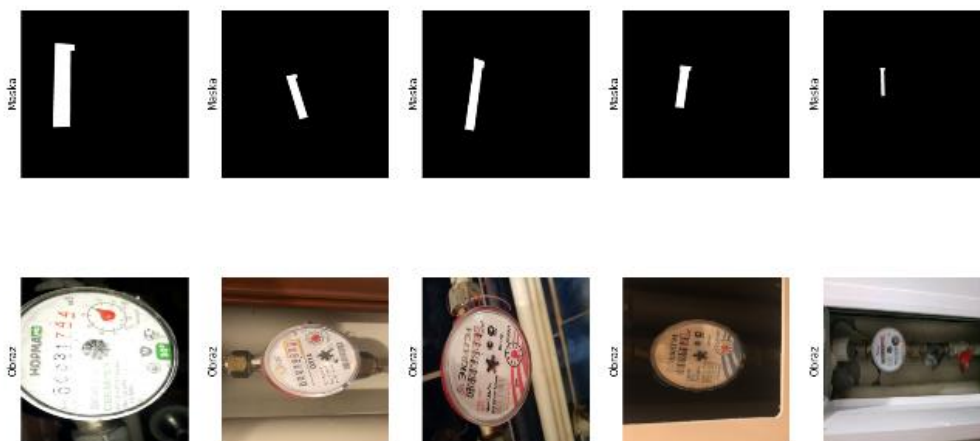
skalujemy do 512×512 z interpolacją najbliższego sąsiada,

```
mask = cv2.resize(mask, dsize=(512, 512), interpolation=cv2.INTER_NEAREST)
```

a następnie konwertujemy na tensor float32 o kształcie [1, H, W]

```
mask = torch.from_numpy(mask.astype(np.float32))[None, ...]
```

Przykład wczytanych danych



Kamień milowy 3: Wybór i implementacja modelu AI

Następnie implementacja modelu U-Net, składającego się enkodera, dekodera, bottleneck'a i klasy zbiorczej. Architektura U-Net opiera się na strukturze kodera-dekodera, w której koder stopniowo kompresuje informacje do reprezentacji o niższym wymiarze. Następnie dekodery dekodują te informacje z powrotem do oryginalnego wymiaru obrazu.

Opis działania modelu:

1. Encoder (downsampling, kompresja):
 - enc1 bierze obraz wejściowy wodomierza (dla RGB w 3 kanałach) i wyciąga z niego cechy. Czyli konwulsja, normalizacja tych cech, wycięcie ujemnych wartości (szumu).
 - pool1 zmniejsza rozdzielczość o połowę.
 - enc2, pool2, enc3, pool3 – powtarzamy to samo ale z coraz większą liczbą filtrów (16 -> 32 -> 64).
2. Bottleneck:
 - Najgłębszy poziom, po nim nie ma już podziału a przechodzimy do upsamplingu.
3. Decoder (upsampling, dekompresja), celem jest uzyskanie globalnego kontekstu:
 - F.interpolate() rozciąga obraz do wcześniejszej rozdzielczości.
 - Torch.cat() dodaje "po chłopsku" ze sobą dane z enkodera i dekodera.
 - Konwulsja -> normalizacja -> ReLU na połączonych cechach.
 - Powtarzamy te kroki w dec3, dec2, dec1.
4. Final (ostatnia warstwa):
 - Dajemy 1 kanał wynikowy (dla segmentacji binarnej) czyli mapę z maską segmentacyjną.

Do przetestowania modelu został użyty plik **train.py**, w którym obraz wejściowy jest przepuszczany przez sieć neuronową (model U-Net). Proces ten składa się z kilku kluczowych etapów:

1. W pierwszej kolejności jest uruchamiany plik przygotowujący strukturę katalogów i walidację danych (plik z wykonaniem drugiego kamienia milowego).

```
# Prepare data
prepare_script = os.path.join(os.path.dirname(__file__), 'prepareDataset.py')
subprocess.run( args: [sys.executable, prepare_script], check=True)
```

2. Następnie, na początku każdej epoki model jest przetaczany w tryb treningowy (**model.train()**). Oznacza to, że:
 - a. **warstwy Dropout są aktywne** – losowo "wyłączają" niektóre neurony, co zapobiega przeuczeniu,
 - b. **warstwy BatchNorm** aktualizują swoje statystyki (średnia i wariancja) na podstawie bieżących danych,
 - c. **liczone są gradienty i aktualizowane wagi modelu**, czyli następuje faktyczne „uczenie się”.

```

for epoch in range(numEpochs):
    model.train() # Set model to training mode (activates Dropout, updates BatchNorm)
    runningLoss = 0.0

    # Iterate over the entire training dataset in batches
    for images, masks in trainLoader:
        images = images.to(device)
        masks = masks.to(device)
        # Clear gradients from the previous step
        optimizer.zero_grad()
        # Forward pass through the network
        outputs = model(images)
        # Calculate loss between model predictions and ground truth masks
        loss = criterion(outputs, masks)
        # Backpropagation: compute gradients
        loss.backward()
        # Update model weights based on gradients
        optimizer.step()
        runningLoss += loss.item()

    # Compute average training loss for this epoch
    avgTrainLoss = runningLoss / len(trainLoader)

```

3. Po zakończeniu etapu treningowego, model przełączany jest w tryb ewaluacyjny (`model.eval()`). W tym trybie:
 - a. **Dropout jest wyłączony**, więc wszystkie neurony są aktywne,
 - b. **BatchNorm używa zapamiętanych (stałych) statystyk** z treningu, a nie obliczanych na bieżąco,
 - c. **gradienty nie są liczone** (oszczędność czasu i pamięci) – model tylko "przewiduje", nie uczy się dalej.

```

# Set model to evaluation mode (disables Dropout, uses running stats in BatchNorm)
model.eval()
runningLoss = 0.0
# Inference loop - same as training, but without gradient tracking
with torch.no_grad():
    for images, masks in valloader:
        images = images.to(device)
        masks = masks.to(device)
        outputs = model(images)
        loss = criterion(outputs, masks)
        runningLoss += loss.item()

# Compute average validation loss for this epoch
avgValLoss = runningLoss / len(valLoader)
print(f"Epoch {epoch + 1}/{numEpochs} - Train Loss: {avgTrainLoss:.4f} - Val Loss: {avgValLoss:.4f}")

```

4. Następnie wykonywana jest walidacja, czyli podobna pętla przejścia przez dane. Używane są dane, których model **nigdy wcześniej nie widział** (nie występują w zbiorze treningowym), co pozwala sprawdzić, **czy model generalizuje** – czyli czy potrafi

przewidzieć poprawnie dla nowych, nieznanych przypadków, a nie tylko zapamiętać dane treningowe. Po każdej epoce porównywane są dwie wartości:

- a. **Train Loss** - błąd na zbiorze treningowym (czyli jak dobrze model uczył się znanych danych),
- b. **Val Loss** - błąd na zbiorze walidacyjnym (czyli jak dobrze model radzi sobie z nowymi danymi).

Jeśli oba błędy maleją równomiernie, to znaczy, że model **uczy się efektywnie i nie przeucza się**. Jeżeli Train Loss spada, a Val Loss rośnie – to znak **przeuczenia** (overfitting): model zapamiętuje dane, ale nie potrafi uogólniać.

```
Epoch 1/25 - Train Loss: 0.4346 - Val Loss: 0.3982
Epoch 2/25 - Train Loss: 0.3714 - Val Loss: 0.3513
Epoch 3/25 - Train Loss: 0.3364 - Val Loss: 0.3206
Epoch 4/25 - Train Loss: 0.3066 - Val Loss: 0.2957
Epoch 5/25 - Train Loss: 0.2802 - Val Loss: 0.2683
Epoch 6/25 - Train Loss: 0.2556 - Val Loss: 0.2460
Epoch 7/25 - Train Loss: 0.2332 - Val Loss: 0.2257
Epoch 8/25 - Train Loss: 0.2129 - Val Loss: 0.2029
Epoch 9/25 - Train Loss: 0.1941 - Val Loss: 0.1871
Epoch 10/25 - Train Loss: 0.1772 - Val Loss: 0.1674
Epoch 11/25 - Train Loss: 0.1613 - Val Loss: 0.1544
Epoch 12/25 - Train Loss: 0.1470 - Val Loss: 0.1401
Epoch 13/25 - Train Loss: 0.1337 - Val Loss: 0.1256
Epoch 14/25 - Train Loss: 0.1207 - Val Loss: 0.1119
Epoch 15/25 - Train Loss: 0.1077 - Val Loss: 0.1009
Epoch 16/25 - Train Loss: 0.0957 - Val Loss: 0.0902
Epoch 17/25 - Train Loss: 0.0855 - Val Loss: 0.0800
Epoch 18/25 - Train Loss: 0.0766 - Val Loss: 0.0721
Epoch 19/25 - Train Loss: 0.0687 - Val Loss: 0.0643
Epoch 20/25 - Train Loss: 0.0616 - Val Loss: 0.0584
Epoch 21/25 - Train Loss: 0.0554 - Val Loss: 0.0513
Epoch 22/25 - Train Loss: 0.0498 - Val Loss: 0.0476
Epoch 23/25 - Train Loss: 0.0447 - Val Loss: 0.0428
Epoch 24/25 - Train Loss: 0.0402 - Val Loss: 0.0383
Epoch 25/25 - Train Loss: 0.0361 - Val Loss: 0.0345
```

W naszym przypadku zarówno Train Loss, jak i Val Loss systematycznie malały z każdą epoką, co wskazuje na skuteczne uczenie się modelu bez oznak przeuczenia i dobrą zdolność generalizacji.

W ramach swegoistego dodatku, wyświetlono szczegóły przebiegu uczenia modelu za pomocą funkcji `summary(model, input_size=(3, 512, 512))`

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 512, 512]	448
BatchNorm2d-2	[-1, 16, 512, 512]	32
ReLU-3	[-1, 16, 512, 512]	0
MaxPool2d-4	[-1, 16, 256, 256]	0
Conv2d-5	[-1, 32, 256, 256]	4,640
BatchNorm2d-6	[-1, 32, 256, 256]	64
ReLU-7	[-1, 32, 256, 256]	0
MaxPool2d-8	[-1, 32, 128, 128]	0
Conv2d-9	[-1, 64, 128, 128]	18,496
BatchNorm2d-10	[-1, 64, 128, 128]	128
ReLU-11	[-1, 64, 128, 128]	0
MaxPool2d-12	[-1, 64, 64, 64]	0
Conv2d-13	[-1, 128, 64, 64]	73,856
BatchNorm2d-14	[-1, 128, 64, 64]	256
ReLU-15	[-1, 128, 64, 64]	0
Conv2d-16	[-1, 64, 128, 128]	110,656
BatchNorm2d-17	[-1, 64, 128, 128]	128
ReLU-18	[-1, 64, 128, 128]	0
Conv2d-19	[-1, 32, 256, 256]	27,680
BatchNorm2d-20	[-1, 32, 256, 256]	64
ReLU-21	[-1, 32, 256, 256]	0
Conv2d-22	[-1, 16, 512, 512]	6,928
BatchNorm2d-23	[-1, 16, 512, 512]	32
ReLU-24	[-1, 16, 512, 512]	0
Conv2d-25	[-1, 1, 512, 512]	17
Total params: 243,425		
Trainable params: 243,425		
Non-trainable params: 0		
Input size (MB): 3.00		
Forward/backward pass size (MB): 364.00		
Params size (MB): 0.93		
Estimated Total Size (MB): 367.93		

Jako wynik funkcji, widzimy zmiany modelu w każdej warstwie (-1,-2,-3, ...). Dla każdej warstwy możemy sprawdzić jaki typ operacji był wykonany:

- **Conv2d** (Warstwa konwolucyjna) – wykrywa cechy (np. krawędzie, tekstury)
- **BatchNorm2d** (Normalizuje dane) – przyspiesza i stabilizuje uczenie
- **ReLU** (Funkcja aktywacji) – dodaje nieliniowość
- **MaxPool2d** (Zmniejsza rozdzielczość obrazu) – agreguje informacje

Szczegóły jej wyniku w formacie **[batch size, liczba kanałów, wysokość, szerokość]**

- **-1** - placeholder dla rozmiaru batcha. Może być dowolny (np. 4, 8, 16)
- **16** - liczba kanałów wyjściowych (np. filtrów konwolucyjnych)
- **512, 512** - rozmiar obrazu wyjściowego (szerokość x wysokość)

Oraz ilość parametrów które były trenowane (**Kolumna Param #**).

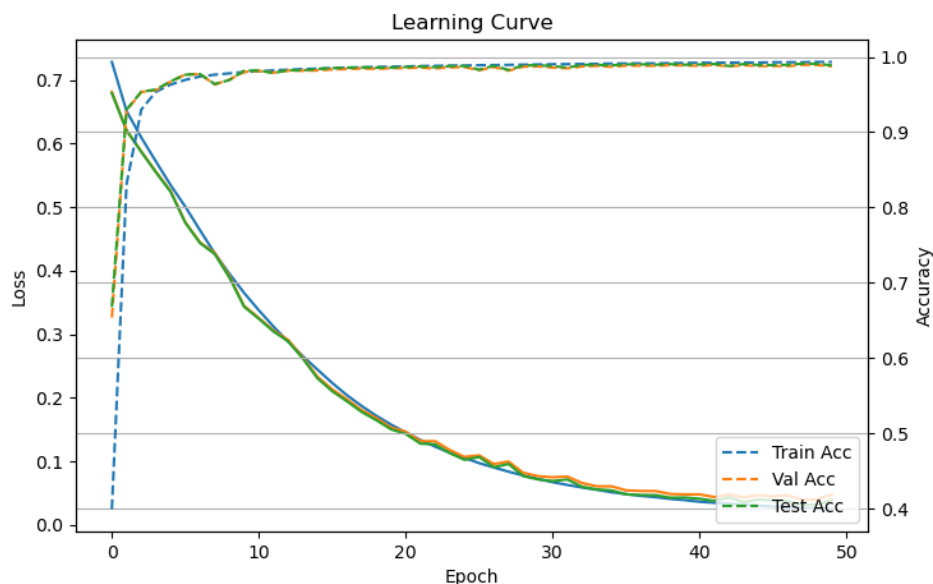
Reszta parametrów:

- **Total params:** 243,425 - łączna liczba parametrów w modelu
- **Trainable params:** 243,425 - wszystkie są uczone przez optymalizator
- **Non-trainable params:** 0 - brak zablokowanych warstw (np. zamrożonych)
- **Input size :** 3.00 MB - Obraz wejściowy zajmuje 3 MB
- **Forward/backward pass size:** 364.00 MB - Tyle pamięci potrzebuje model w czasie przejścia w przód i do obliczenia gradientów
- **Params size:** 0.93 MB - Parametry modelu (wagi, biasy) zajmują mniej niż 1 MB
- **Total size:** ~368 MB - Szacowane zużycie pamięci RAM/GPU przy jednym batchu obrazu 512×512

Kamień milowy 4: Ocena wyników modelu i optymalizacja

Po zakończonym treningu na zbiorze treningowym dokonaliśmy próby na zbiorze testowym:

- Zebrano oraz narysowano funkcje strat i dokładności w zależności od minionej epoki. Przez to możemy zobaczyć czy nasz model nadal się uczy, stoi lub się przeucza



Możemy na powyższym wykresie zaobserwować:

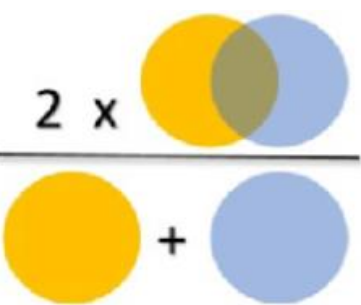
- Szybki spadek strat i wzrost accuracy w pierwszych 5–10 epokach. Już od 1–2 epoki widzimy gwałtowny spadek trainLoss (niebieska ciągła) z ~0.7 do ~0.3, równocześnie trainAcc (niebieska przerywana) skacze z ~0.93 do ~0.98. Podobnie dla zbiorów walidacyjnego (valLoss pomarańczowa, valAcc przerywana pomarańczowa) i testowego (testLoss zielona, testAcc przerywana zielona). To oznacza, że model szybko uczy się rozróżniać tło od obiektu, bez znacznego przeuczenia.
- Stabilizację metryk. Po około 10 epokach wszystkie krzywe strat zaczynają się wypłaszczać, docierając pod koniec do wartości ~0.02–0.03. Dokładność zbliża się do ~0.992 i również przestaje się znacząco poprawiać.
- Brak nadmiernego przeuczenia (overfittingu). Straty i accuracy na zbiorach treningowym, walidacyjnym i testowym idą niemal identycznie przez cały przebieg. Gdyby model przeuczał się do trainu, widzielibyśmy rosnący odstęp pomiędzy trainLoss a valLoss (lub wzrost trainAcc przy malejącym valAcc). Tutaj tego nie ma.

Wnioski i kolejne kroki

Model osiąga bardzo niską stratę i wysoką pikselową dokładność, co jest jednak w dużej mierze efektem silnego niezrównoważenia klas (~99 % pikseli to tło). Dlatego accuracy traktujemy jedynie jako uzupełnienie. Kluczowe metryki dla segmentacji to Dice i IoU, które ustabilizowały się odpowiednio na poziomie ~0.73 i ~0.64 co jest dobrym wynikiem, zwłaszcza dla tak wymagającego zbioru. Z kolei Hausdorff przyjął wartość nieskończoności, ponieważ w niektórych próbkach model przewidział pustą maskę (brak pikseli „1”), co powoduje nieokreśloność w obliczeniu maksymalnej odległości między zestawami punktów.

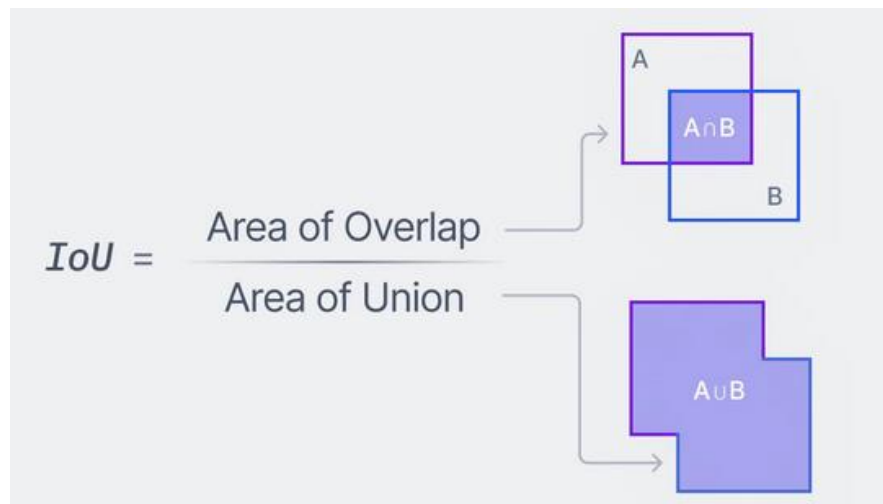
```
Test Dice:      0.7273
Test IoU:       0.6447
Test Hausdorff: inf
```

Dice Coefficient mierzy dwukrotną względną wielkość pokrycia predykcji i GT, czyli jak bardzo ich obszary się pokrywają, z wyraźniejszym naciskiem na obszar wspólny. Jest to miara często stosowana w segmentacji medycznej.

$$\text{Dice Score} = \frac{2 \times \text{Intersection}}{\text{S}_{\text{Algorithm}} + \text{S}_{\text{GroundTruth}}}$$


```
# Dice coefficient
def dice_coeff(pred, target, smooth=1e-6): 1 usage  ↗ vvszewczyk *
    pred = pred.flatten()
    target = target.flatten()
    intersection = (pred * target).sum() # 2*|pred ∩ GT| # GT - Ground Truth
    return (2. * intersection + smooth) / (pred.sum() + target.sum() + smooth) # 2*|pred ∩ GT|
```

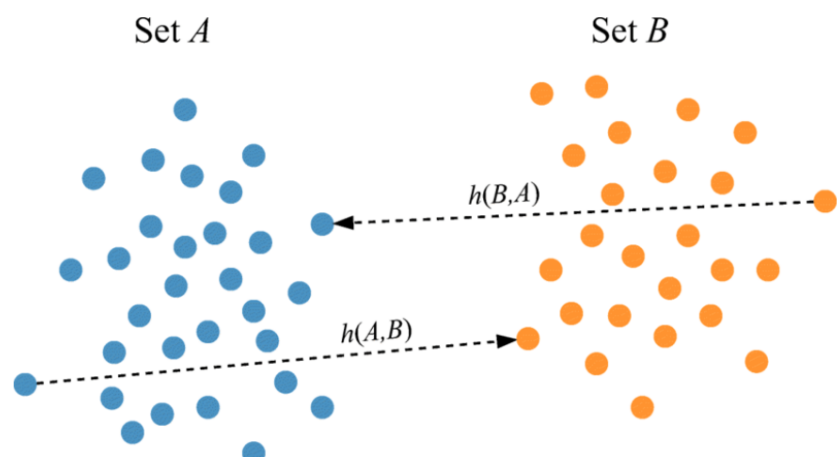
Intersection over Union (IoU) mierzy stosunek wspólnego obszaru predykcji i maski prawdziwej (GT) do ich sumy minus część wspólna, czyli jak duży procent obszaru obiektu został poprawnie odgadnięty. Dodatkowo mamy stałą smooth, dla uniknięcia dzielenia przez 0 i dziwnych wyników



```
# Intersection over Union
def iou_coeff(pred, target, smooth=1e-6): 1 usage vvszewczyk *
    pred = pred.flatten()
    target = target.flatten()
    intersection = (pred * target).sum() # |pred ∩ GT|
    union = pred.sum() + target.sum() - intersection # |pred| + |GT| - |pred ∩ GT|
    return (intersection + smooth) / (union + smooth) # smooth to avoid division by 0
```

Hausdorff Distance mierzy największą minimalną odległość między punktami dwóch zbiorów punktów (tutaj pikseli maski i predykcji). Ocenia, jak daleko najgorszy („najbardziej odległy”) punkt przewidywanej obwódki jest od prawdziwej obwódki.

```
p_pts = np.argwhere(p.squeeze()==1)
m_pts = np.argwhere(m.squeeze()==1)
hd1 = directed_hausdorff(p_pts, m_pts)[0]
hd2 = directed_hausdorff(m_pts, p_pts)[0]
hausdorff_dists.append(max(hd1, hd2))
```



Dodatkowo, do poprawy modelu przydała nam się dokładność pikseli (**Pixel-wise accuracy**). Jest to stosunek liczby pikseli poprawnie sklasyfikowanych ($\text{pred} == \text{target}$) do całkowitej liczby pikseli w obrazie.

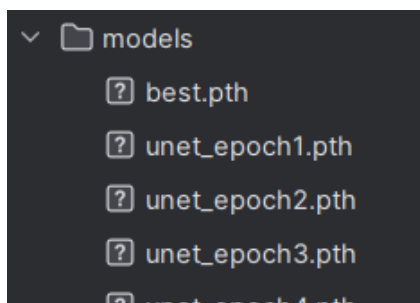
```
# Pixel-wise accuracy
def pixel_accuracy(pred, target):
    return (pred == target).mean()
```

Ocena w jaki sposób można poprawić dotychczas stworzone rozwiązanie:

- Rozbudowa bloku konwolucyjnego. Zamiast pojedynczej konwolucji w każdym etapie, użyć DoubleConv (dwie konwolucje + bottleneck + ReLU(funkcja aktywacji $\max(0, x)$)), zwiększyć bazową liczbę filtrów (np. z 32 na 64) i dorzucić Dropout2d, by podnieść pojemność sieci i zapobiec przeuczeniu.
- Data augmentations. Losowe obroty, przycięcia, zmiany jasności/kontrastu, elastic deformation takie jak rozciągnięcia, co zwiększy różnorodność treningowego zbioru i wzmocni odporność modelu.

Kamień milowy 5: Wdrożenie modelu i monitorowanie

Po zakończeniu treningu każdy stan sieci jest zapisywany jako checkpoint, a najlepszy model według walidacji – jako best.pth w katalogu models/. Dzięki temu w dowolnym momencie można wznowić trening lub uruchomić inferencję bez ponownego uczenia od zera. Poniżej przedstawiono najlepsze uzyskane wyniki ze zdjęć z zestawu zdjęć:



Image



GT Mask



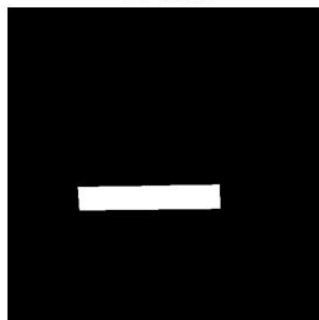
Predicted Mask



Image



GT Mask



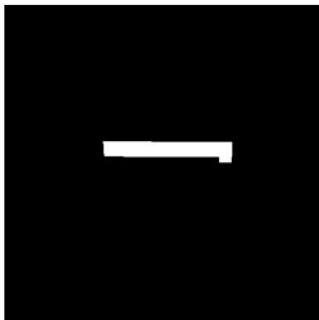
Predicted Mask



Image



GT Mask



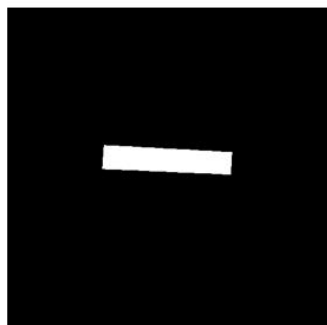
Predicted Mask



Image

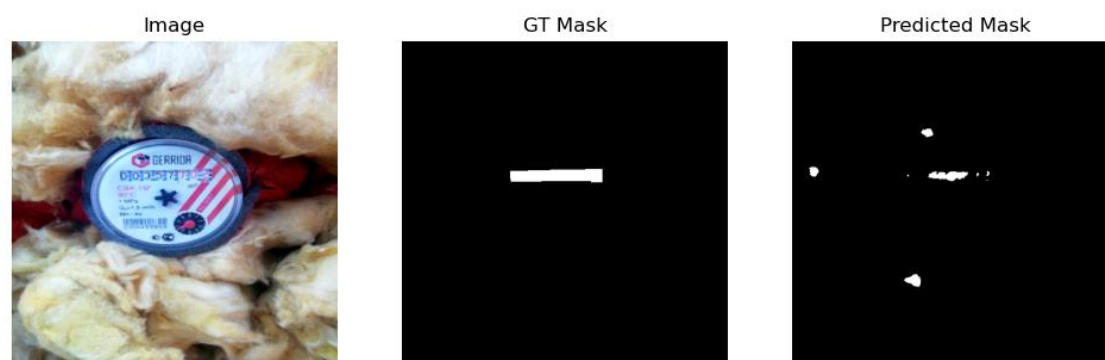
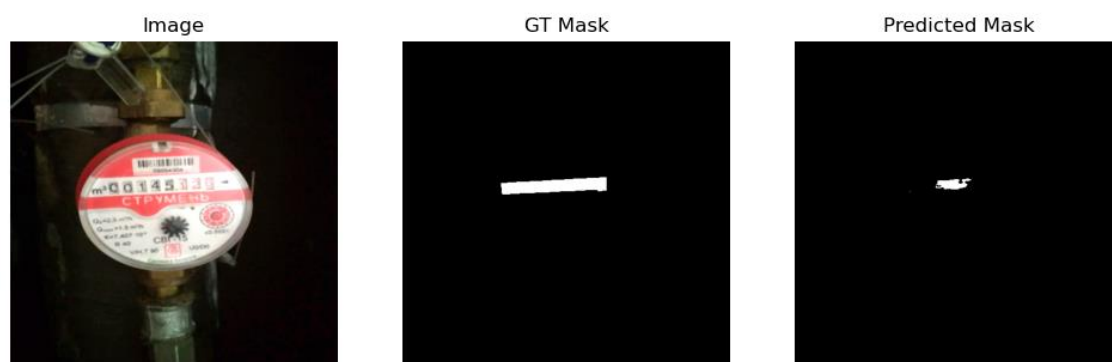
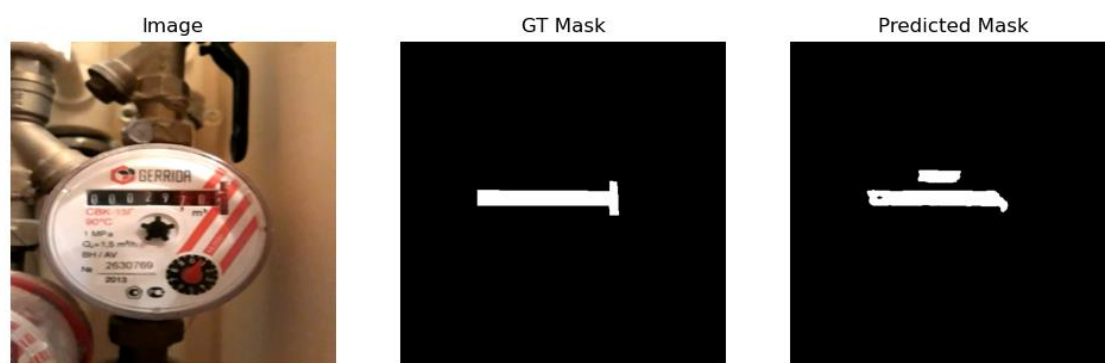
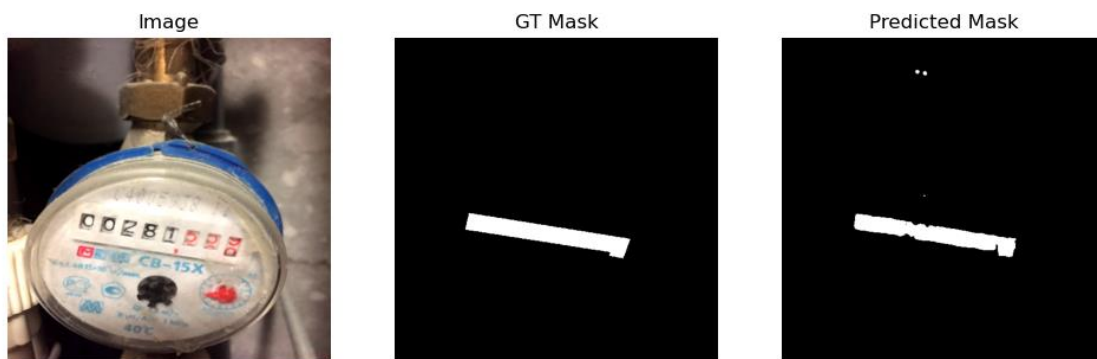


GT Mask



Predicted Mask





Następnie przeprowadzono testy działania modelu na zdjęciach wykonanych samodzielnie, czyli nie mającymi żadnego powiązania z zestawem „Water Meters”. Dzięki temu możliwe było sprawdzenie rzeczywistej zdolności modelu do generalizacji w “warunkach bojowych”:

Original



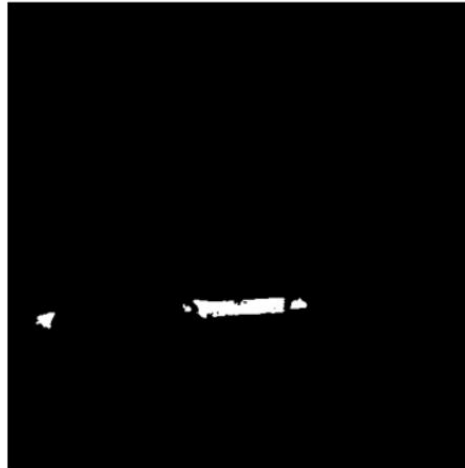
Predicted mask



Original



Predicted mask



Original



Predicted mask



Original



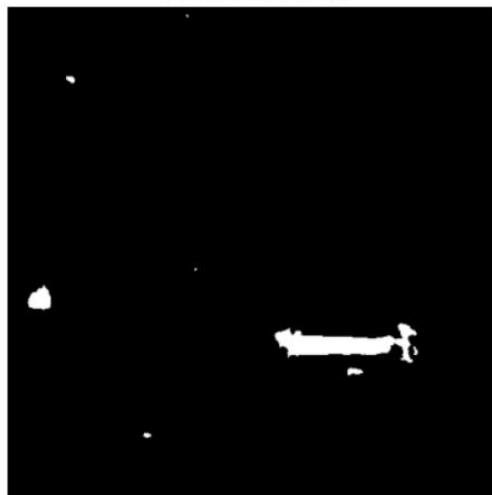
Predicted mask



Original



Predicted mask



Original



Predicted mask



Original



Predicted mask



Wyniki inferencji są zadowalające. Model, pomimo drobnych pomyłek w przypadku zdjęć gorszej jakości, generalnie był w stanie skutecznie zidentyfikować miejsce występowania licznika na zdjęciach zarówno z Internetu, jak i wykonanych własnoręcznie, co potwierdza jego użyteczność w praktyce.

Podsumowanie całego projektu

W ramach projektu opracowano model segmentacji obrazów wodomierzy oparty na sieci U-Net. W kamieniu milowym 1 określono temat, cel i wymagania, a także przygotowano środowisko pracy i zbiór danych. W kamieniu milowym 2 przetworzono obrazy i maski, znormalizowano dane oraz podzielono je na zbiory treningowy, walidacyjny i testowy. W kamieniu milowym 3 zaimplementowano i przetrenowano zmodyfikowany (jedna konwolucja i jedno ReLU) model U-Net, zapewniając jego poprawne działanie i unikając przeuczenia. W kamieniu milowym 4 przeprowadzono dokładną ocenę modelu za pomocą metryk segmentacyjnych (Dice, IoU, Hausdorff), potwierdzając jego skuteczność i stabilność. W kamieniu milowym 5 zapisano najlepszy model, przeprowadzono inferencję na zdjęciach spoza zestawu danych i potwierdzono zdolność modelu do generalizacji w praktycznych zastosowaniach.