SIGK2025Z
Projekt 2
Mapowanie tonalne

Oskar Nowak i Paulina Staszewska

17 listopada 2025

1 Cel projektu

Celem projektu było stworzenie mapowania tonalnego, które wkorzystuje sieci neuronowe oraz przeprowadzenie nad nim eksperymentów.

2 Zbiór danych

Zbiór danych to SI-HDR, w którym wykorzystano pliki z folderu $reference$ oraz odpowiadające im pliki z folderu $clip\_97$ według nazwy. Zbiór został podzielony na następujące podzbiory:

- treningowy (70%, czyli 127 plików z każdego folderu),
- walidacyjny (10%, czyli 18 plików z każdego folderu),
- testowy (20%, czyli 36 plików z każdego folderu).

W obu folderach było w sumie po 181 plików.

3 Model

Architektura sieci składa się z 3 koderów, fusion blocku i dekodera (listing 1). Każdy z tych elementów posiada:

- kodery:
  - wejściem do każdego z kodera to obraz 3 kanałowy
  - 3 warstwy konwolucji z *kernel* i *paddingem* równymi kolejno 3 i 1,
  - po każdej tych warstw użyta jest funckja ReLU,
  - w ostatniej warstwie otrzymywana jest mapa cech z ilością kanałów równą 64
- fusion block:
  - łączy wyjścia koderów w jedno,
  - zmniejsza liczbę kanałów do 64 (krok dodany dodatkowo ze względu na zbyt duże wykorzystanie pamięci RAM w systemie)
  - 2 warstwy konwolucji
- dekoder:
  - 3 warstwy konwolucji z *paddingem* równymi 1,
  - po każdej tych warstw użyta jest funkcja ReLU, prócz ostatniej, po której występuje funkcja sigmoid

– ilość kanałów odpowiada ilości kanałów z kodera w odwróconej kolejności

– wyjściem dekodera jest obraz o 3 kanałach

```
# -------------------- NETWORK --------------------
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3,16,3,padding=1)
        self.conv2 = nn.Conv2d(16,32,3,padding=1)
        self.conv3 = nn.Conv2d(32,64,3,padding=1)
    def forward(self,x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        return x

class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(64,32,3,padding=1)
        self.conv2 = nn.Conv2d(32,16,3,padding=1)
        self.conv3 = nn.Conv2d(16,3,3,padding=1)
    def forward(self,x, img1,img2,img3):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.conv3(x)
        x = torch.sigmoid(x + img1 + img2 + img3)
        return x

class TMONet(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()
        # fusion convs
        self.fusion_conv1 = nn.Conv2d(64*3, 64, 3, padding=1) # zmniejszone ze
            wzgl du na zbyt duze uzycie RAM w porowaniu do oryginalnej arhcitektury
        self.fusion_conv2 = nn.Conv2d(64, 64, 1)

    def forward(self, i0, i1, i2):
        # encode each input
        o0 = self.encoder(i0)
        o1 = self.encoder(i1)
        o2 = self.encoder(i2)

        # concat features
        o0 = torch.cat([o0, o1, o2], dim=1) # nadpisywanie dla optymalizacji
            wykorzystania RAM
        o1 = o2 = None

        # fusion block
        o0 = F.relu(self.fusion_conv1(o0))

        o0 = self.fusion_conv2(o0)

        # decode
        o0 = self.decoder(o0, i0, i1, i2)
        return o0
```

Listing 1: Architektura sieci

3.1 Funkcja straty

Sieć korzysta z funkcji straty o nazwie *fcm_loss*, która korzysta z *feature contrast masking loss*, *VGG*, lokalnej średniej i odchylonej standardowej oraz maskowania (listing 2). Strata ta bierze pod uwagę percepcji wzrokowej człowieka.

```
# ---------------------------
# VGG feature extractor
```

```
vgg_model = models.vgg19(pretrained=True).features.to(device).eval()
for p in vgg_model.parameters():
    p.requires_grad = False

def vgg_features(x, layers=['0','5','10']):  # odpowiada VGG11,21,31 w TF
    features = []
    h = x
    for idx, layer in enumerate(vgg_model):
        h = layer(h)
        if str(idx) in layers:
            features.append(h)
    return features  # lista tensorow

# ----------------------------
# Gaussian kernel
_gaussian_cache = {} # optymalizacja aby raz wczytac
def gaussian_kernel(size=13, sigma=2.0, channels=3, device='cpu'):
    key = (size, sigma, channels)
    if key in _gaussian_cache:
        return _gaussian_cache[key].to(device)
    coords = np.arange(size) - size//2
    x, y = np.meshgrid(coords, coords)
    kernel = np.exp(-(x**2 + y**2)/(2*sigma**2))
    kernel = kernel / kernel.sum()
    kernel = torch.tensor(kernel, dtype=torch.float32, device=device)
    kernel = kernel.view(1,1,size,size).repeat(channels,1,1,1)
    _gaussian_cache[key] = kernel
    return kernel

def local_mean_std(x, kernel_size=13, sigma=2.0):
    C = x.shape[1]
    w = gaussian_kernel(kernel_size, sigma, C).to(x.device)
    x_pad = F.pad(x, (kernel_size//2,)*4, mode='reflect')
    mean_local = F.conv2d(x_pad, w, groups=C)
    mean_sq = F.conv2d(x_pad**2, w, groups=C)
    std_local = torch.sqrt(torch.clamp(mean_sq - mean_local**2, min=1e-8))
    return mean_local, std_local

# ----------------------------
# Feature contrast masking
def sign_num_den(x, gamma=0.5, beta=0.5, sigma=2.0, kernel_size=13):
    mean_local, std_local = local_mean_std(x, kernel_size, sigma)
    norm_num = torch.sign(x - mean_local) * torch.abs((x - mean_local)/(mean_local.abs
        ()+1e-8))**gamma
    norm_den = 1.0 + (std_local / (mean_local.abs()+1e-8))**beta
    return norm_num, norm_den

def feature_contrast_masking(x, gamma=0.5, beta=0.5, sigma=2.0, kernel_size=13):
    num, den = sign_num_den(x, gamma, beta, sigma, kernel_size)
    return num / den

def masking_loss(pred, target, gamma=0.5, beta=0.5, sigma=2.0, kernel_size=13):
    f_pred = feature_contrast_masking(pred, gamma=1.0, beta=beta, sigma=sigma,
        kernel_size=kernel_size)
    f_target = feature_contrast_masking(target, gamma=gamma, beta=beta, sigma=sigma,
        kernel_size=kernel_size)
    return F.l1_loss(f_pred, f_target)

# ----------------------------
# FCM loss
def fcm_loss(pred, target, gamma=0.5, beta=0.5, sigma=2.0, kernel_size=13):
    feats_pred = vgg_features(pred, layers=['0','5','10'])   # odpowiada VGG11,21,31
    with torch.no_grad():
      feats_target = vgg_features(target, layers=['0','5','10'])
    loss_total = 0.0
    for f_pred, f_target in zip(feats_pred, feats_target):
        loss_total += masking_loss(f_pred, f_target, gamma=gamma, beta=beta, sigma=
            sigma, kernel_size=kernel_size)
    loss_total /= len(feats_pred)
```

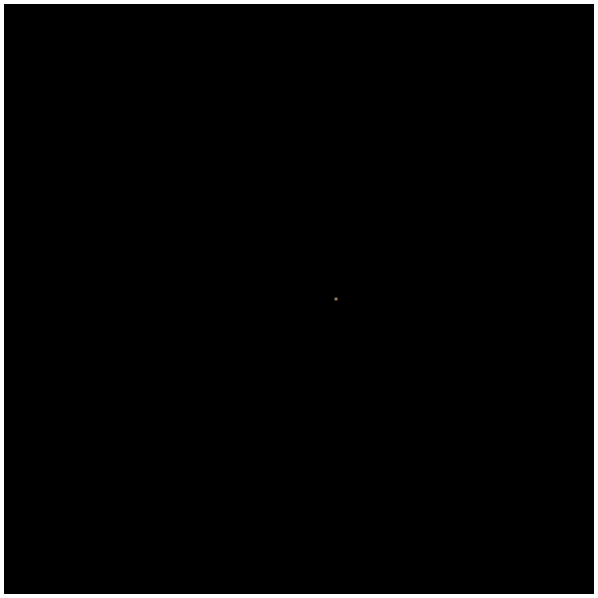
```
68      return loss_total
```

Listing 2: Funkcja straty

Model został wytrenowany według następujących parametrów:

- rozmiar batcha: 1,
- ilość epok: 49,
- funkcja straty: fcm_loss,
- optymalizator: Adam
- learing rate: 2^{-5}
- checkpoint: $model\_11\_17\_16\_1\_14\_48.pth$.

4 Wizualizacja wyników



Rysunek 1: Obraz 156_Mantiuk.png otrzymany metodą Mantiuk

Na podstawie zdjęcie z folderu *clip_97* (rysunek 4) można zauważyć, że obrazy otrzymany metodą Mantiuk (rysunek 1) jest wręcz czarny. Natomiast obrazy otrzymane przez sieć neurnową (rysunek 3) i metodę Reinhard (rysunek 2) są prawie podobne do siebie.

5 Wyniki

Ze zbioru wybrano 20% plików i na podstawie ich obliczono wartości metryk, które zostały zaprezentowano w tabeli 1. Z niej warto podkreślić, że wynik idealny równy 1 dla SSIM dla metody *input_97* to

Metoda	SSIM	BRISQUE
input_97	1,000	18,757
Reinhard_Operator	0,839	37,427
Mantiuk_Operator	0,779	82,894
Learned_Operator	0,661	19,747

Tabela 1: Metryki

Rysunek 2: Obraz 156_Reinhard.png otrzymany metodą Reinhard

tak naprawdę punkt odniesienia na podstawie, którego wykonano porównanie otrzymanych obrazów SDR, które zostały zmapowane z HDR. Według współczynnika SSIM najlepszą metodą jest metoda Reinhard, która uzyskała wartość równą 0,839. Kolejno najlepszą metodą była wykorzystująca sposobu Mantiuk. Model z siecią neurnonową uzyskał najgorszy wynik z pośród wszystkich sposobów, a patrząc jedynie w obrębie współczynnika SSIM jest to wynik średni. Natomiast według metryki BRISQUE najlepszy wynik osiągnęła metoda wykorzystująca sieć neuronową. Średnią metodą jest metoda Reinharda. Najgorszy wynik otrzymała metoda Mantiuk. Biorąc pod uwagę metrykę BRISQUE to model z siecią neurnonową otrzymuje wartość zbliżoną do baseline.

Wszystkie otrzymane zdjęcia z modelu w fazie treningu zapisano do foldera o nazwie *results*, a w folderze o nazwie $results/TEST$ są pliki stworzone w czasie testowania. W folderze o nazwie *compare* są zdjęcia otrzymane z pozostałych metod oraz pliki HDR. Foldery te są umieszczone w folderze dla danego modelu, którego one dotyczą.

6 Eksperymenty

W projekcie początkowo werjsa modelu podczas treningu miała następujące parametry:

- rozmiar batcha: 1,
- ilość epok: 50,
- funkcja straty: fcm_loss,
- optymalizator: Adam
- learing rate: 2^{-4}
- checkpoint: $model\_11\_16\_4\_25\_47\_49.pth$.

Finalna wersja modelu posiada learing rate równe 2^{-5} ze względu na to, że większa wartość powodowała zbyt duże wahania w otrzymywaniu wag sieci, a cały trening był niestabilny.

Pierowotna wersja model nie wykonywała ona zmniejszania obrazów do 255 na 255 z interpolacją ustawioną na $INTER\_AREA$. Osiągnięte wartości metryk tego modelu zostały umieszczone w tabeli 2.

Na podstawie 2 wersji modelu można powiedzieć, że ostateczna wersja modelu uzyskuje lepsze wyniki pod względem współczynnika BRISQUE. Natomiast, biorąc pod uwagę parametr SSIM to dla metody Mantiuk wypada jednie lepiej nowsza wersja modelu niż starsza. Jednak orginalny model uzyskał wartość BRISQUE znacznie gorszą od od swojej nowszej wersji i od baseline.

Rysunek 3: Obraz 156.png otrzymany wyuczonym modelem

Metoda	SSIM	BRISQUE
input_97	1,000	19,657
Reinhard_Operator	0,716	55,721
Mantiuk_Operator	0,837	103,632
Learned_Operator	0,633	57,798

Tabela 2: Metryki

7 Wnioski

Zaimplementowana sieć neuronowa pod względem metryki BRISQUE daje najlepszy rezultat, prawie wręcz porównywalny z orginałem pod względem precepcji widzenia człowieka. Jednak, jeśli chodzi o SSIM to model zwraca obrazy, które są średniej jakości m. in. pod względem struktury i szumu. Finalna wersja modelu jest lepsza niż jej pierwowzór w kwesti mapowania pliku HDR na SDR oraz w procesie trenowania.

Rysunek 4: Obraz 156_SDR_input97.png - baseline