

# Życie na krawędzi - sprawozdanie

- Przy wyborze obrazów kierowałem się swoją pasją do fantastyki, więc wybrałem zdjęcie wydarzenia, które nigdy nie nastąpiło (przynajmniej według niektórych):



- Przekształcenie obrazu do grayscale'a:

```
image = Image.open('./mro.png')
image_tensor = T.ToTensor()(image)
image_tensor = image_tensor.unsqueeze(0)

conv1x1 = nn.Conv2d(3, 1, kernel_size=1, stride=1, padding=0, bias=False)
weights = torch.tensor([0.3, 0.5, 0.1]).view(1, 3, 1, 1)
conv1x1.weight.data = weights
output = conv1x1(image_tensor)

output_image = output.squeeze().detach().numpy()
output_image = Image.fromarray((output_image * 255).astype('uint8'))
output_image.save('grayscale_result.png')
```

Przez godzinę walczyłem z komplikacją tego skryptu (zapomniałem o batch dimension, a potem próbowałem jakkolwiek wyświetlić wynik konwolucji), padding i stride są tak dobrane, żeby nie zmieniać rozmiaru obrazu. Wynik wyszedł nie najgorszy:



3. Użyłem poolingu max, jako że chcemy odwzorować krawędzie, więc intuicyjnie ‘average’ kojarzy się z rozmyciem.

```
pooling = nn.MaxPool2d(kernel_size=4)
output = pooling(output)
output_image = output.squeeze().detach().numpy()
output_image = Image.fromarray((output_image * 255).astype('uint8'))
output_image.save('pooling_result.png')
```



4. Rozmycie gaussowskie, nauczony wstępem do tego zadania oraz podpierając się poradnikiem do zajęć, udało się przejść nawet sprawnie. Filtr o rozmiarze 5 sprawdził się całkiem nieźle:

```
def gaussian_kernel(size, sigma=1):
    size = int(size) // 2
    x, y = np.mgrid[-size:size+1, -size:size+1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g = np.exp(-(x**2 + y**2) / (2.0*sigma**2)) * normal
    return g

gauss_blur = nn.Conv2d(1, 1, kernel_size=5, stride=1, padding=2, bias=False)
weights = torch.tensor(gaussian_kernel(5)).float().view(1, 1, 5, 5)
gauss_blur.weight.data = weights
output = gauss_blur(output)

output_image = output.squeeze().detach().numpy()
output_image = Image.fromarray((output_image * 255).astype('uint8'))
output_image.save('gaus_blur_result.png')
```



5. Liczymy oba kanały z macierzy Sobela, wyznaczając gradient i jego kierunek:

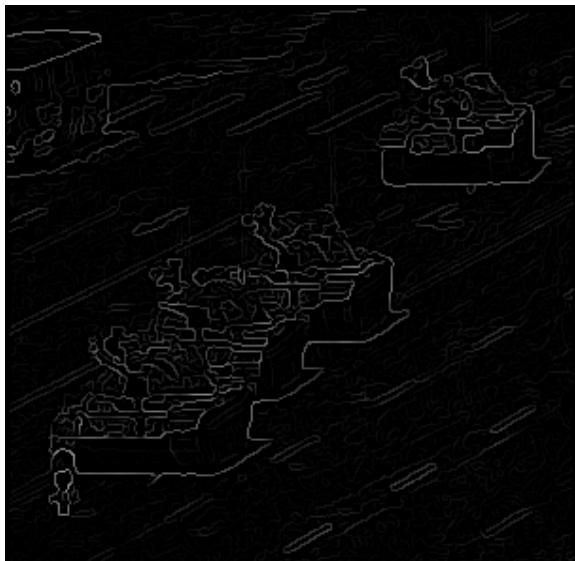
```
sobel_conv = nn.Conv2d(1, 2, kernel_size=3, stride=1, padding=1, bias=False)
weights = torch.tensor([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], [[1, 2, 1], [0, 0, 0], [-1, -2,
-1]]).float().view(2, 1, 3, 3)
sobel_conv.weight.data = weights
output = sobel_conv(output)

x = output[:, 0, :, :]
y = output[:, 1, :, :]
euklides = torch.sqrt(torch.pow(x, 2) + torch.pow(y, 2))
euklides = euklides / euklides.max() * 255
arctan = torch.arctan2(y, x)

output_image = euklides.squeeze().detach().numpy()
output_image = Image.fromarray((output_image).astype('uint8'))
output_image.save('gradient_result.png')
```



6. Przechodzimy do kroku “Non Maximum Supression”. Po przeczytaniu treści podpunktu i napotkanych trudnościach, pożyczyłem implementację z tutoriala:



7. Threshold ustawilem na poziomie 50, bo powyżej czołgi były trochę już niewyraźne, a poniżej 30 za bardzo było widać randomowe szramy na asfalcie (bez podwójnego thresholda to trochę taka ciuciubabka, czy uwzględnimy za dużo nieistotnych krawędzi, czy pominiemy te istotniejsze). Na koniec normalizacja, żeby krawędzie były wyraźniejsze:



8. Na koniec wyskalowanie w górę (UpSample + pikselowy padding) i porównanie z oryginałem:



Wyszło całkiem średnio, czyli świetnie. Bardzo przyjemny lab, aczkolwiek jak się z Pythonem nie pracuje, to idzie utknąć na syntaxie bibliotek 😞