Raport z zadania 3. - Generative Adversarial Networks

Michał Stefanik

1 listopada 2023

1 Wstęp

Do wykonania zadania używałem języka Python 3.10.13 z biblioteką PyTorch 2.1.0.

2 Model dyskryminatora

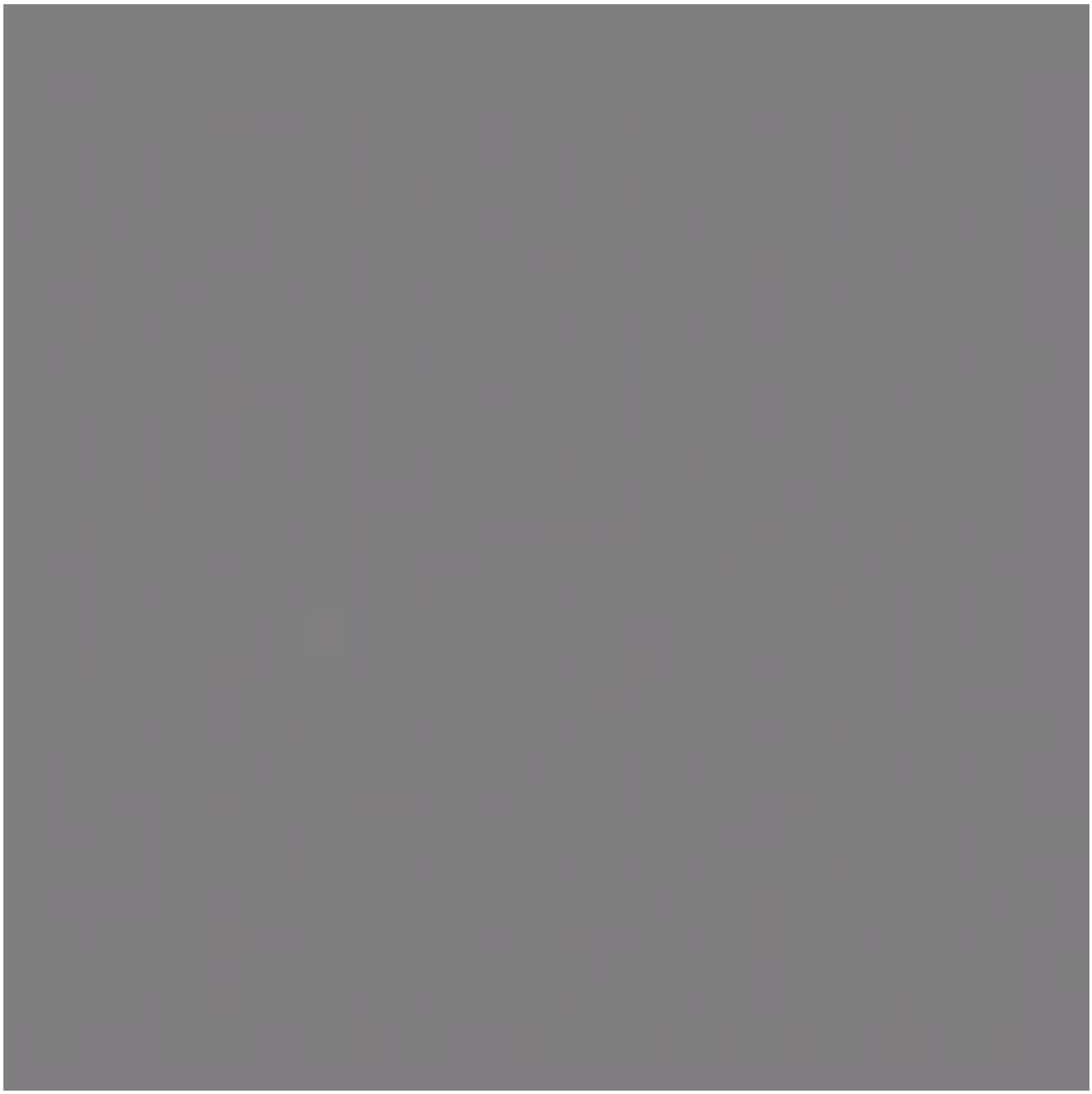
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=4, stride=2, padding=2)
        self.norm1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=2)
        self.norm2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=2)
        self.norm3 = nn.BatchNorm2d(128)
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(3200, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.norm1(x)
        x = F.leaky_relu(x, 0.2)
        x = self.conv2(x)
        x = self.norm2(x)
        x = F.leaky_relu(x, 0.2)
        x = self.conv3(x)
        x = self.norm3(x)
        x = F.leaky_relu(x, 0.2)
        x = self.flatten(x)
        x = F.dropout(x, 0.5)
        x = self.linear1(x)
        x = F.sigmoid(x)
        return x
```

3 Model generatora

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.linear1 = nn.Linear(64, 64 * 4 * 4)
        self.conv1 = nn.ConvTranspose2d(64, 64, kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.ConvTranspose2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.conv3 = nn.ConvTranspose2d(128, 256, kernel_size=4, stride=2, padding=1)
        self.conv4 = nn.Conv2d(256, 3, kernel_size=5, stride=1, padding=2)

    def forward(self, x):
        x = self.linear1(x)
```

```
        x = x.view(-1, 64, 4, 4)
        x = self.conv1(x)
        x = F.leaky_relu(x, 0.2)
        x = self.conv2(x)
        x = F.leaky_relu(x, 0.2)
        x = self.conv3(x)
        x = F.leaky_relu(x, 0.2)
        x = self.conv4(x)
        x = torch.tanh(x)
        return x
```

Bez żadnego treningu, generator generuje obrazki przypominające szum. Przykład takigo obrazka wygenerowanego z użyciem wektorów o wartościach z rozkładu normalnego standardowego na rysunku 1.



Rysunek 1: Przykładowy obrazek wygenerowany przez generator bez treningu

4 Zbiór danych

Zbiór danych to dostarczone przez prowadzącego zdjęcia ciast marchewkowych. Przykładowe zdjęcia zostały pokazane na rysunku 2. Zdjęcia zostały przeskalowane do rozmiary 32 x 32, a ich wartości przeskalowane do zakresu [-1, 1].

Rysunek 2: Przykładowe zdjęcia ciast marchewkowych

5 Przygotowanie się do działania z optimizerem

Żeby mieć pewność, że optimizer będzie zmieniał tylko wagi dyskryminatora, a nie generatora, zrobiłem mały ekspertyment. Na listingu 1 umieściłem kod prostego modelu z dwoma warstwami. Następnie stworzyłem optimizer, który będzie optymalizował tylko pierwszą warstwę. Po wykonaniu jednego i więcej kroków optymalizacji, wagi drugiej warstwy nie zmieniły się. Wiemy więc, że optimizer zmienia jedynie wagi, które mu podamy jako parametr.

Listing 1: Eksperyment z optimizerem

```
class Minimodel(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.lin = nn.Linear(2, 2)
        self.lin2 = nn.Linear(2, 2)

    def forward(self, x):
        x = self.lin(x)
        x = self.lin2(x)
        return x


minimodel = Minimodel()
input_val = torch.randn(10, 2)
loss_fn = lambda x: ((x - 42) ** 2).sum()
optimizer = torch.optim.SGD(minimodel.lin.parameters(), lr=0.01)

pred = minimodel(input_val)
loss = loss_fn(pred)
loss.backward()
optimizer.step()
```

Po wykonaniu kilku takich kroków wartość loss spadła znacząco, a wartości wyniku modelu zbliżyły się do 42.

6 Trening

Do treningu zostały użyte następujące hiperparametry:

Listing 2: Parametry treningu

```
discriminator = Discriminator().to(device)
generator = Generator().to(device)

lrd = 0.00001
lrg = 0.000015
betas = (0.5, 0.999)

discriminator_optimizer = torch.optim.Adam(
    discriminator.parameters(), lr=lrd, betas=betas
)
generator_optimizer = torch.optim.Adam(
    generator.parameters(), lr=lrg, betas=betas
)
```

6.1 Trening dyskryminatora

Trening dyskryminatora składa się z dwóch części. W pierwszej części na wejściu podajemy mu prawdziwe obrazki, które oznaczamy jako prawdziwe. W drugiej części podajemy mu obrazki wygenerowane przez generator, które oznaczamy jako fałszywe. Po zebraniu gradientu z obu części, wykonujemy jeden krok optymalizacji wag dyskryminatora.

```
# train discriminator on true images
discriminator_optimizer.zero_grad()
pred1 = discriminator(image_batch)
target = torch.ones((pred1.shape[0], 1), device=device)
loss1 = F.binary_cross_entropy(pred1, target)
loss1.backward()

# train discriminator on generated images
pred2 = discriminator.forward(
    generator.forward(torch.randn(16, 64, device=device))
)
target = torch.zeros((pred2.shape[0], 1), device=device)
```

```
loss2 = F.binary_cross_entropy(pred2, target)
loss2.backward()

discriminator_optimizer.step()
disc_losses.append((loss1 + loss2).item()/2)
```

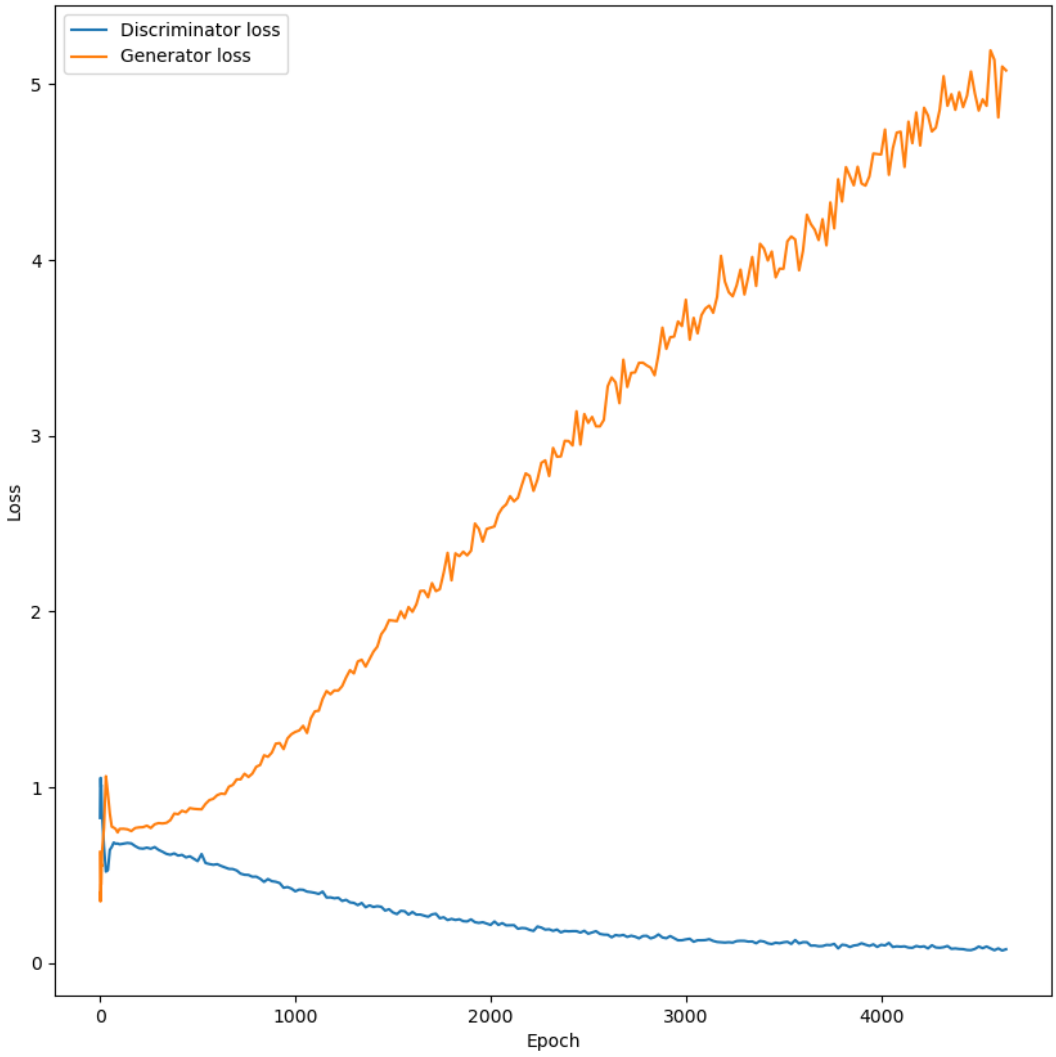
6.2 Trening generatora

Trening generatora polegał na podaniu wygenerowanych przez niego obrazków do dyskryminatora i oznaczeniu ich jako prawdziwe. Następnie zebraniu gradientu i wykonaniu jednego kroku optymalizacji wag generatora.

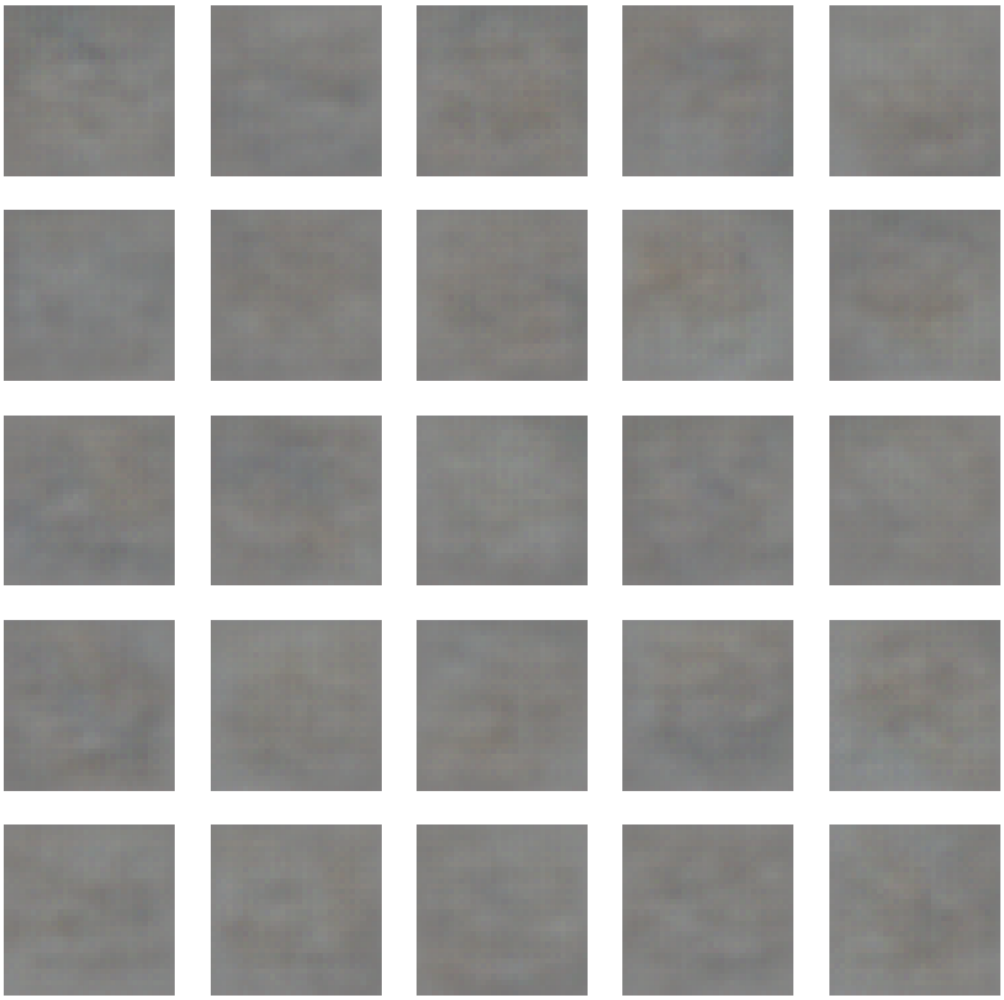
```
# train generator
generator_optimizer.zero_grad()
batch = generator.forward(torch.randn(16, 64, device=device))
pred = discriminator(batch)
target = torch.ones((pred.shape[0], 1), device=device)
loss = F.binary_cross_entropy(pred, target)
loss.backward()
generator_optimizer.step()
gen_losses.append(loss.item())
```

6.3 Monitoring modelu

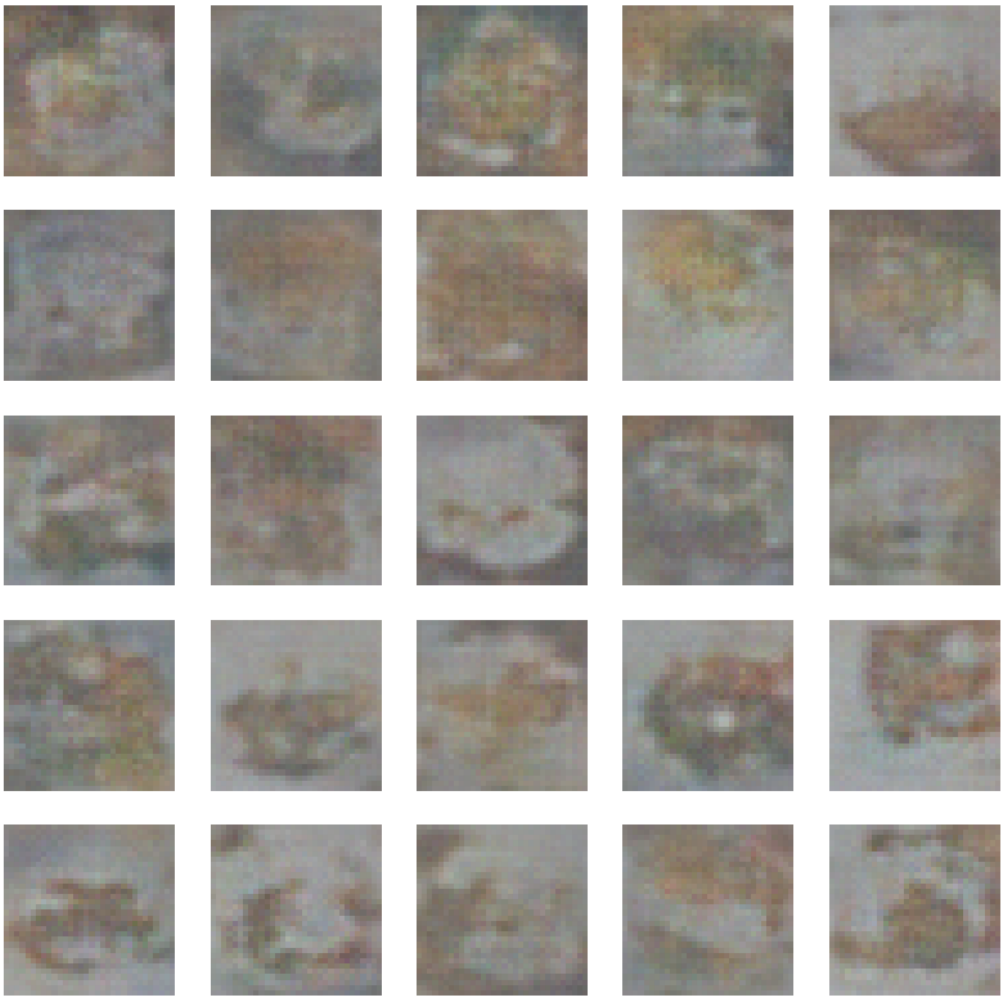
W celu monitorowania modelu co kilkadziesiat epok zapisywałem średnią wartość funkcji loss dla dyskryminatora i generatora. Wyniki zostały pokazane na rysunku 3. Oprócz kosztu zapisywałem też obrazki wygenerowane przez generator dla kilku stałych wektorów wejściowych. Zostały one zaprezentowane na rysunkach 4, 5, 6 i 7. Dalszy trening nie przynosił znaczących zmian w wizualnej jakości generowanych obrazków. Wartośći funkcji loss zaczęły się znacząco różnić od siebie.



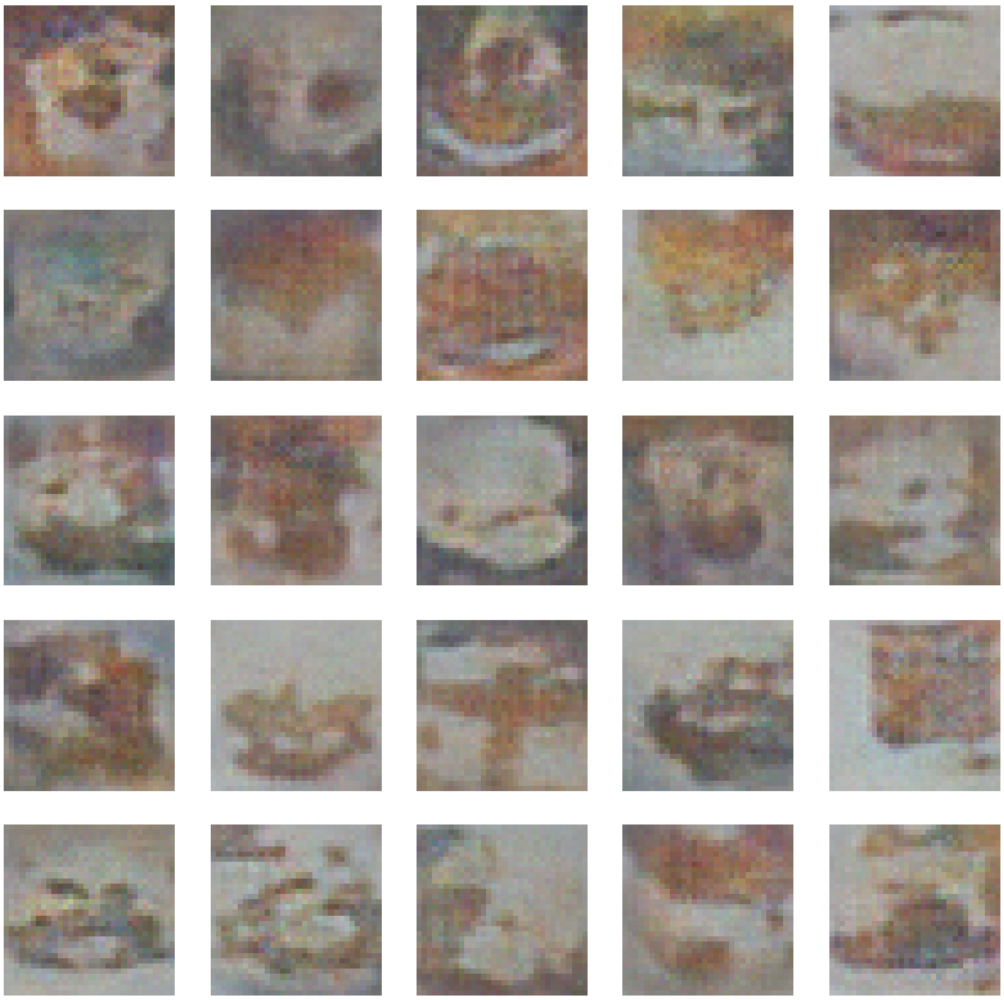
Rysunek 3: Wartości funkcji loss dla dyskryminatora i generatora



Rysunek 4: Obrazki wygenerowane przez generator po 100 epokach



Rysunek 5: Obrazki wygenerowane przez generator po 500 epokach



Rysunek 6: Obrazki wygenerowane przez generator po 1000 epokach

Rysunek 7: Obrazki wygenerowane przez generator po 2000 epokach

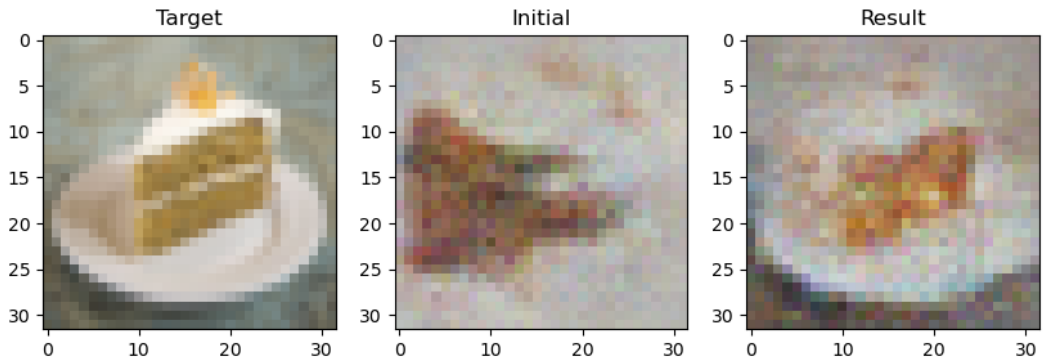
7 Dopasowanie wejściowego wektora do obrazka

Do tego celu użyłem modelu po 2000 epokach. W celu znalezienia wektora wejściowego, obliczyłem gradient funkcji loss względem wektora wejściowego. Wyniki dopasowania zostały pokazane na rysunku 8. Kod optymalizacji znajduje się na listingu 3. Na rysunku 9 została pokazana ewolucja obrazka od przypadkowego do dopasowanego. Na rysunku 10 zostały pokazane obrazki wygenerowane z wektora wejściowego ze zmienianą jedną wartością.

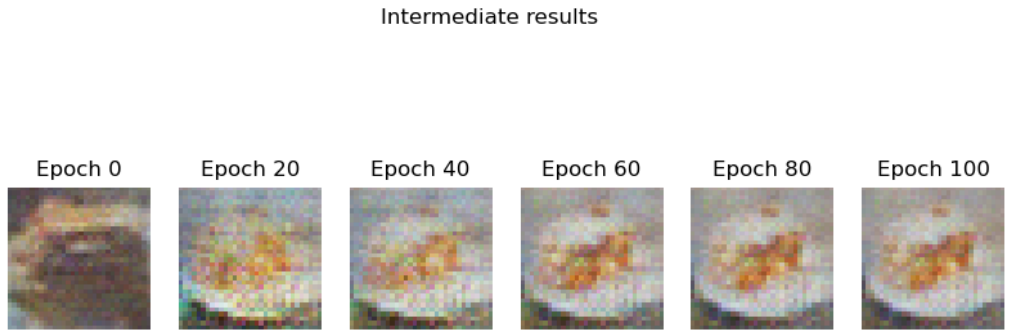
Listing 3: Dopasowanie wektora wejściowego do obrazka

```
input_vector = torch.randn(1, 64, device=device, requires_grad=True)
target_image = true_images_dataset[0]

image_optim = torch.optim.SGD([input_vector], lr=0.01, momentum=0.9)
loss_fn = lambda x: ((x - target_image) ** 2).sum()

for i in range(100):
    image_optim.zero_grad()
```

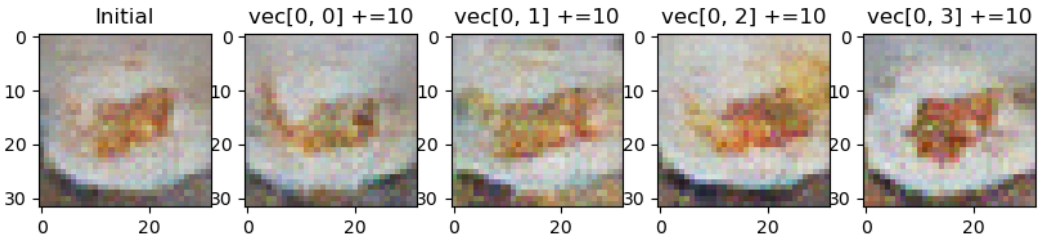
```
result = generator.forward(input_vector)
loss = loss_fn(result)
loss.backward()
image_optim.step()
```



Rysunek 8: Dopasowanie wektora wejściowego do obrazka

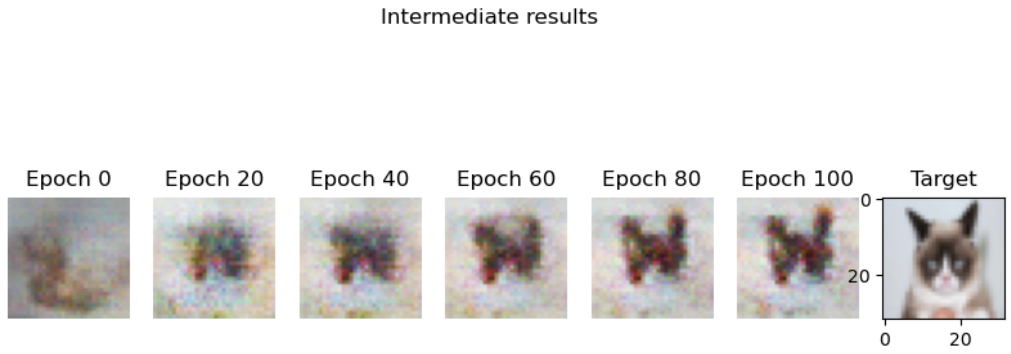


Rysunek 9: Ewolucja obrazka od przypadkowego do dopasowanego



Rysunek 10: Obrazki wygenerowane z wektora wejściowego ze zmienianą jedną wartością

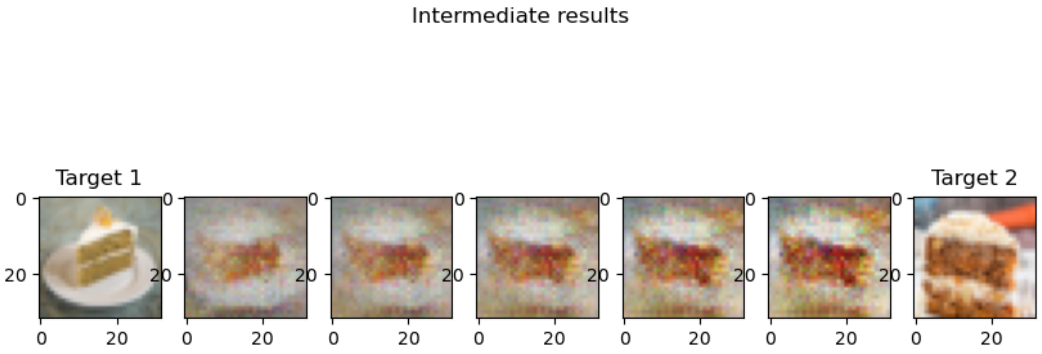
Dodatkowo procedura ta została powtórzona dla innego obrazka (grumpy cat), niezwiązanego z ciastem marchewkowym. Wyniki zostały pokazane na rysunku 11.



Rysunek 11: Dopasowanie wektora wejściowego do innego obrazka

8 Interpolacja między obrazkami

Ostatnim zadaniem było wygenerowanie interpolacji między dwoma obrazkami. W tym celu do dwóch obrazków zostały dopasowane wektory wejściowe. Następnie wygenerowano kilka obrazków leżących na linii w hiperpłaszczyźnie rozpiętej na wektorach wejściowych. Wyniki zostały pokazane na rysunku 12.



Rysunek 12: Interpolacja między obrazkami

9 Wnioski i uwagi do zadania

Zadanie na pewno satysfakcjonujące. Udało się wygenerować obrazki, które jak się dobrze przypatrzy to przypominają ciasta marchewkowe. Niestety nie udało się wygenerować obrazków, które byłyby jak dla mnie dobre jakościowo. Dobór hiperparametrów był na pewno kluczowy do uzyskania dobrych wyników, na co zeszło sporo czasu. Wartości funkcji loss dla dyskryminatora i generatora zaczęły się znacząco różnić od siebie, co przyznam, że mnie zaskoczyło. Starałem się to ograniczyć przez zmniejszenie współczynnika uczenia dyskryminatora, pomogło wydłużyć równowagę, ale nie całkowicie. Najdłużej trwało uczenie - w moim przypadku co najmniej 8 godzin, a fakt jakiegokolwiek uczenia było widać dopiero po co najmniej godzinie. Wyniki nie są tak piękne jak te przykładowe, ale może to być kwestia błędu w implementacji architektury sieci lub procesie uczenia.