Zadanie 3.

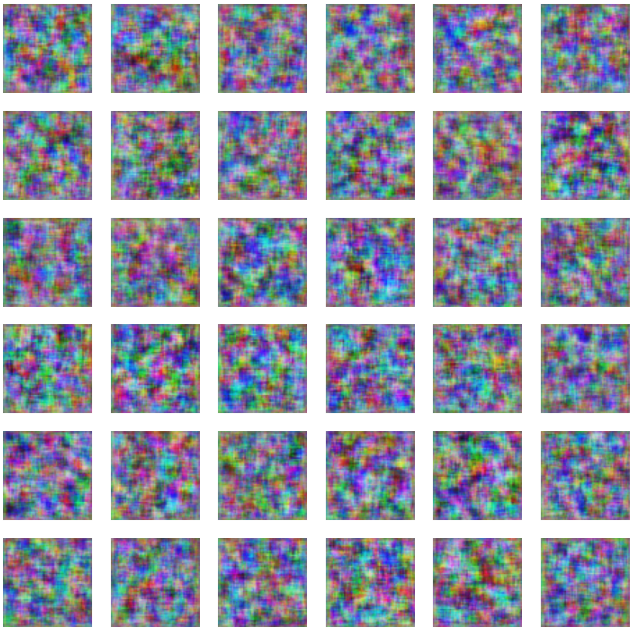
Zadanie wykonałem korzystając z frameworka PyTorch. Od początku wykorzystywałem obrazy o rozmiarach 64x64 zamiast 32x32, dlatego też odpowiednie parametry w blokach konwolucyjnych generatora i dyskryminatora są zmienione względem podanych w instrukcji. Poniżej zamieściłem definicję modelu dyskryminatora. Dyskryminator nie używa na końcu funkcji sigmoidalnej, gdyż jako funkcji kosztu używam wariantu BCEloss z PyTorcha, która przyjmuje wartości logitów.

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1)
        self.conv2 = nn.Conv2d(64, 64, 4, 2, 1)
        self.conv3 = nn.Conv2d(64, 64, 4, 2, 1)
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(64)
        self.lin = nn.Linear(64 * 8 * 8, 1)

    def forward(self, x: Tensor) -> Tensor:
        x = F.leaky_relu(self.bn1(self.conv1(x)), negative_slope=0.2)
        x = F.leaky_relu(self.bn2(self.conv2(x)), negative_slope=0.2)
        x = F.leaky_relu(self.bn3(self.conv3(x)), negative_slope=0.2)
        x = x.flatten(1)
        x = self.lin(x)
        return x
```

Również ze względu na wykorzystywanie obrazów o rozmiarach 64x64 w modelu generatora zwiększyłem rozmiar wektora zmiennych ukrytych (latent variables) z 64 na 128. Poniżej zamieściłem definicję modelu generatora.

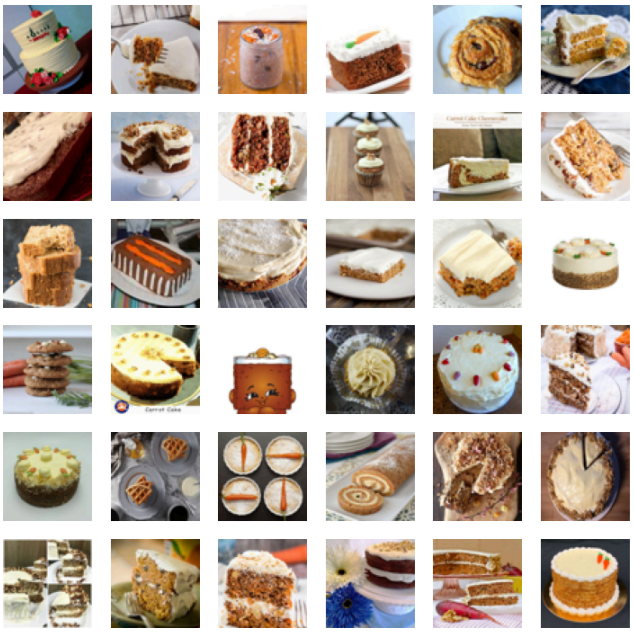
```
class Generator(nn.Module):
    def __init__(self, latent_dim=128):
        super().__init__()
        self.lin = nn.Linear(latent_dim, 64 * 8 * 8)
        self.conv1 = nn.Conv2d(64, 64, 3, 1, 1)
        self.conv2 = nn.Conv2d(64, 128, 3, 1, 1)
        self.conv3 = nn.Conv2d(128, 256, 3, 1, 1)
        self.conv4 = nn.Conv2d(256, 3, 5, 1, 2)
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)

    def forward(self, x: Tensor) -> Tensor:
        x = self.lin(x)
        x = x.reshape(-1, 64, 8, 8)
        x = F.leaky_relu(
              self.bn1(self.conv1(F.interpolate(x, scale_factor=2))), negative_slope=0.2)
        x = F.leaky_relu(
              self.bn2(self.conv2(F.interpolate(x, scale_factor=2))), negative_slope=0.2)
        x = F.leaky_relu(
              self.bn3(self.conv3(F.interpolate(x, scale_factor=2))), negative_slope=0.2)
        x = F.tanh(self.conv4(x))
        return x
```

Poniżej zamieściłem wyjście generatora dla różnych wektorów losowych o wartościach z rozkładu normalnego.



Następnie zająłem się przygotowaniem danych treningowych. Po wczytaniu ich z dysku korzystając z biblioteki Pillow, zmieniłem ich rozmiar na 64x64 oraz znormalizowałem wartości, tak aby były z przedziału od -1 do +1.

```
dirname = "./crawled_cakes"
imgs_dataset = np.array([Image.open(f"{dirname}/{f}").convert("RGB").resize((64, 64))
                         for f in os.listdir(dirname)])
imgs_dataset = imgs_dataset.transpose(0, 3, 1, 2)
imgs_dataset = imgs_dataset.astype(np.float32)
imgs_dataset = (imgs_dataset / 127.5) - 1.0

show(next(make_batch(imgs_dataset, 36)))
```



Fazę uczenia dyskryminatora zaimplementowałem zgodnie z podanym opisem z małą zmianą polegającą na osobnym obliczeniu wartości funkcji straty dla prawdziwych i fałszywych obrazów i następnie ich dodanie, zamiast konkatenacji przykładów z obu zbiorów.

```
        # TRAIN DISCRIMINATOR
        # ============================
        optim_d.zero_grad()

        # Train discriminator with real images
        t_real = torch.ones(batch_size, 1)
        y_real = D(X_real)

        # Train discriminator with fake images
        noise = torch.randn(batch_size, latent_dim)
        X_fake: Tensor = G(noise)

        t_fake = torch.zeros(batch_size, 1)
        y_fake = D(X_fake.detach())

        loss_d = criterion(y_real, t_real) + criterion(y_fake, t_fake)
        loss_d.backward()

        optim_d.step()
```

Analogicznie zaimplementowałem fazę uczenia generatora.

```
        # TRAIN GENERATOR
        # ============================
        optim_g.zero_grad()

        t_adver = torch.ones(batch_size, 1)
        y_adver = D(X_fake)

        loss_g = criterion(y_adver, t_adver)
        loss_g.backward()

        optim_g.step()
```

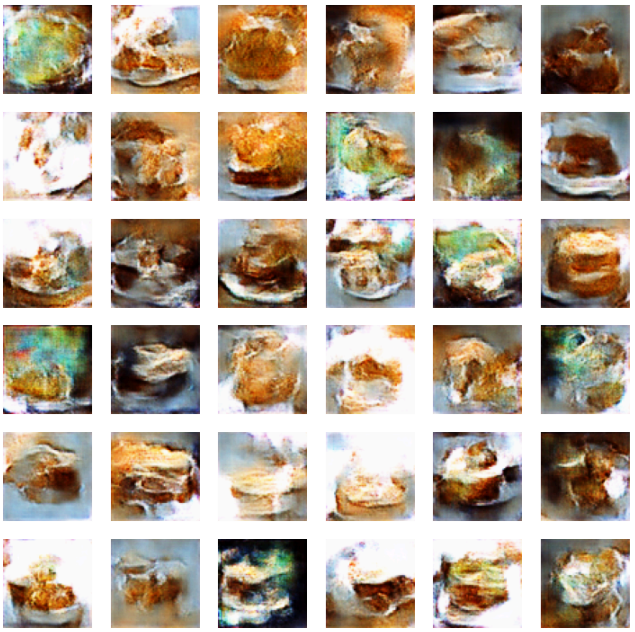
Zaimplementowałem również cykliczne zapisywanie aktualnych parametrów modeli oraz stanu optymalizatorów wraz z zapisem generowanych obrazów dla ustalonego przed pętlą treningową tensora zawierającego 36 wektorów losowych.

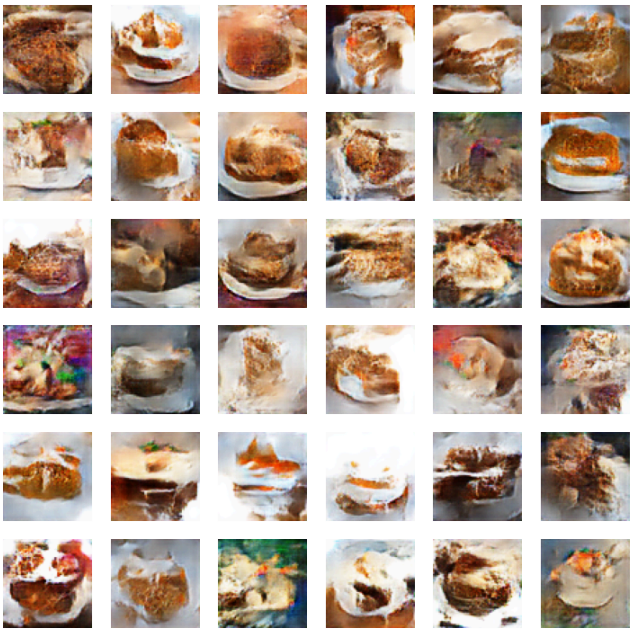
```
    if (epoch + 1) % log_period == 0:
        show(G(fixed_noise).detach(), savepath=f"./outputs/img{epoch+1}.png")
        torch.save(
            {
                "epoch": epoch + 1,
                "G_state_dict": G.state_dict(),
                "D_state_dict": D.state_dict(),
                "optim_g_state": optim_g.state_dict(),
                "optim_d_state": optim_d.state_dict(),
            },
            f"./checkpoints/params_epoch{epoch+1}.pt",
        )
```

Do treningu przyjąłem następujące wartości hiper-parametrów.

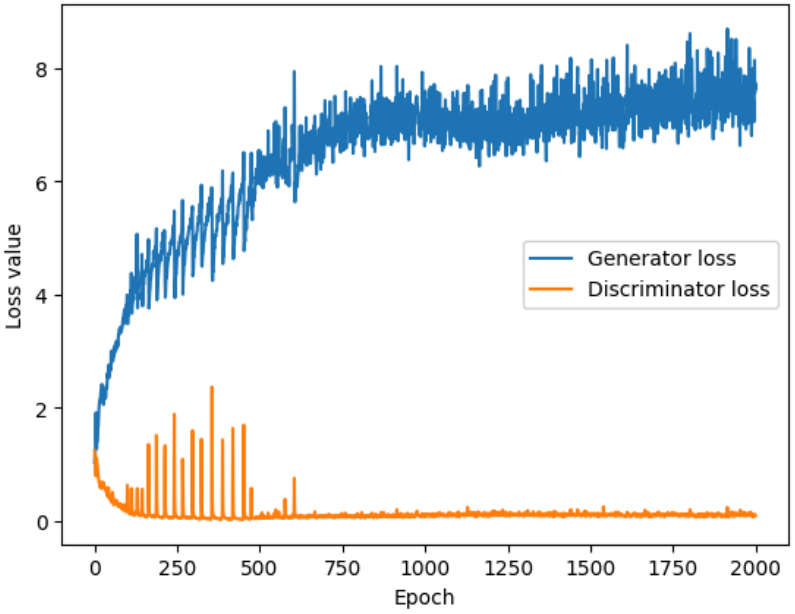
```
lr = 0.0002
betas = (0.5, 0.999)
weight_decay = 1e-2
batch_size = 16
epochs = 2_000
```

Poniżej zamieściłem generowane przez generator obrazy dla ustalonych wektorów losowych po 50, 300 i 500 epokach treningu. Po 500 epokach rezultaty (oglądane z dużej odległości) nawet przypominają wyroby cukiernicze.





Poniżej zamieściłem również wykres wartości funkcji straty dla dyskryminatora i generatora.



Niestety po 500 epokach loss generatora zaczął dość wyraźnie rosnąć i widoczny był spadek jakości generowanych obrazów. Z tego względu do ostatniej części zadania wczytałem archiwalne wagi generatora właśnie po 500 epokach treningu.

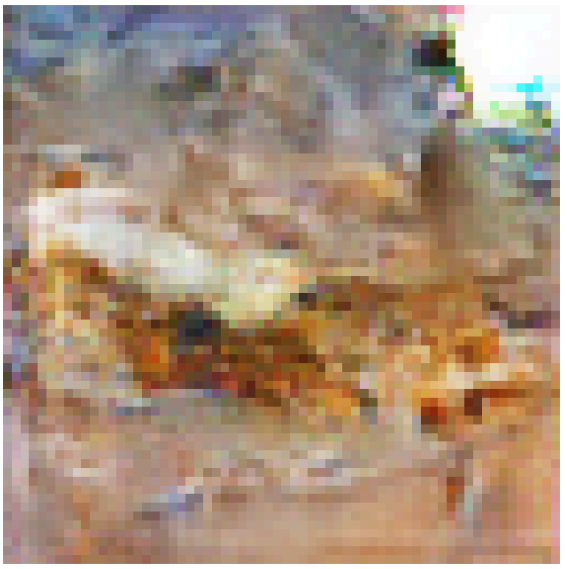
```
params = torch.load("checkpoints/params_epoch500.pt", weights_only=False)
G = Generator()
G.load_state_dict(params["G_state_dict"])
```

Poniższy fragment kodu implementuje pętlę iteracyjnie zmieniającą wejściowy wektor zmiennych ukrytych w taki sposób, aby wyjście generatora imitowało zadany obraz target_img.

```
noise = torch.randn(1, 128)
noise.requires_grad = True
criterion = F.mse_loss
optimizer = optim.SGD([noise], lr=1e-1, momentum=0.99, nesterov=True)

for epoch in (pbar := trange(epochs)):
    optimizer.zero_grad()
    fake_img = G(noise)
    loss = criterion(fake_img, target_img)
    loss.backward()
    optimizer.step()
```

Jako obraz do odtworzenia wybrałem ten przedstawiony poniżej. Obok zamieściłem również uzyskaną imitację po 10 000 epok oraz ciąg pośrednich imitacji dla epok 1, 50, 100, 1000 i 5000.

W drugim eksperymencie wybrałem obraz niezwiązany z danymi treningowymi, przedstawiający samolot. Poniżej zamieściłem obraz docelowy oraz uzyskaną imitację. Uzyskana imitacja była zupełnie niepodobna do docelowego obrazka.

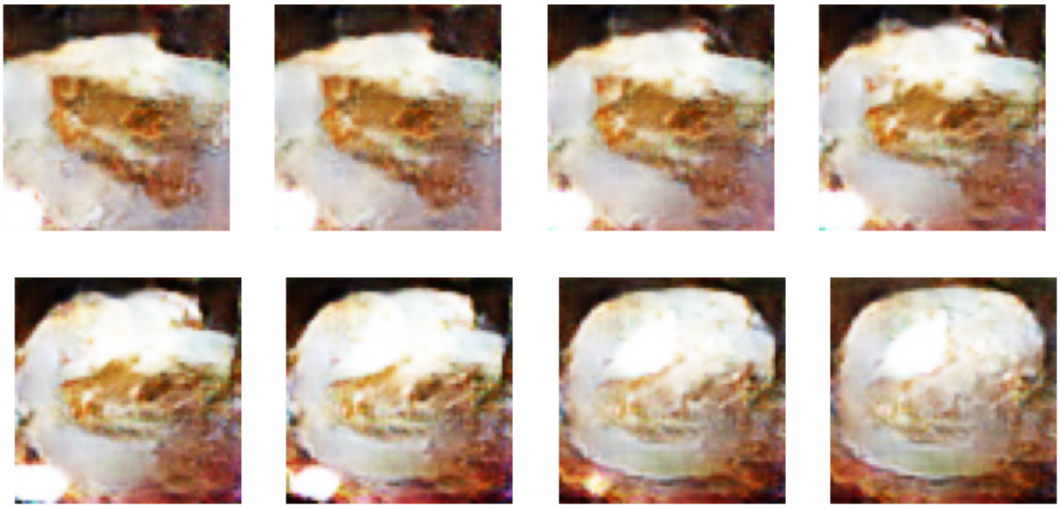


Ostatnią częścią zadania było wybranie innego obrazka ze zbioru danych treningowych, wyznaczenie dla niego odpowiedniego wektora zmiennych losowych korzystając z wyżej zaimplementowanej procedury oraz wygenerowanie obrazów dla wektorów pośrednich między wektorem dla pierwszego i drugiego ciasta. Poniżej zamieściłem wybrany obrazek oraz uzyskaną imitację.

Poniższy fragment kodu implementuje obliczanie i wyświetlanie obrazów pośrednich.

```
latent = lambda alpha: latent_cake_1 + alpha * (latent_cake_2 - latent_cake_1)

fig = plt.figure(figsize=(16, 16))
ncols, nrows, idx = 8, 1, 1

for alpha in np.linspace(0, 1, ncols):
    im: Tensor = G(latent(alpha)).detach()
```

```
    im = (im + 1.0) / 2.0
    fig.add_subplot(nrows, ncols, idx)
    idx += 1
    plt.axis("off")
    plt.imshow(im[0, ...].permute(1, 2, 0).cpu().numpy())
```



Podsumowując całe zadanie należy stwierdzić, iż trudność w treningu zaimplementowanego modelu wynika przede wszystkim z jego niestabilności i dość dużej wrażliwości na wartości hiper-parametrów. W moim przypadku pomimo, iż cały trening modelu (2000 epok) trwał około 1 godziny, to tak naprawdę rezultaty uzyskane po 500 epokach treningu, czyli ok. 15 minutach są już najlepsze, a później ze względu na niestabilność modelu marnujemy tylko zasoby obliczeniowe, gdyż generowane obrazy są coraz gorszej jakości. Pomimo tego uzyskane rezultaty po 500 epokach są całkiem niezłe, w szczególności są porównywalne do tych uzyskanych przez autora zadania.