Metody Rozpoznawania Obrazów
Zadanie 03 - Czy sieci neuronowe śnią o ciastach marchewkowych?

Wojciech Michaluk

07.11.2025

Opis kolejnych etapów zadania

UWAGA: Na przestrzeni całego zadania, w implementacji sieci neuronowych oraz pętli treningowych korzystam z modułów biblioteki *keras*.

Na początku przygotowuję model *dyskryminatora*. Testowałem różne modyfikacje tej sieci (zmieniając liczbę filtrów czy siłę regularyzacji L2) na potrzeby treningu, ale finalnie przyjmuję architekturę tak, jak opisano w poleceniu.

```
discriminator = models.Sequential()
discriminator.add(layers.Input(shape=(32, 32, 3)))
discriminator.add(layers.Conv2D(
    filters=32, kernel_size=(4, 4), strides=(2, 2), padding="same",
    kernel_regularizer=regularizers.L2(0.01)
))

discriminator.add(layers.BatchNormalization())
discriminator.add(layers.LeakyReLU(negative_slope=0.2))
discriminator.add(layers.Conv2D(
    filters=64, kernel_size=(4, 4), strides=(2, 2), padding="same",
    kernel_regularizer=regularizers.L2(0.01)
))

discriminator.add(layers.BatchNormalization())
discriminator.add(layers.LeakyReLU(negative_slope=0.2))
discriminator.add(layers.Conv2D(
    filters=64, kernel_size=(4, 4), strides=(2, 2), padding="same",
    kernel_regularizer=regularizers.L2(0.01)
))

discriminator.add(layers.BatchNormalization())
discriminator.add(layers.LeakyReLU(negative_slope=0.2))
discriminator.add(layers.Flatten())
discriminator.add(layers.Dropout(rate=0.2))
discriminator.add(layers.Dense(units=1, activation='sigmoid'))
```

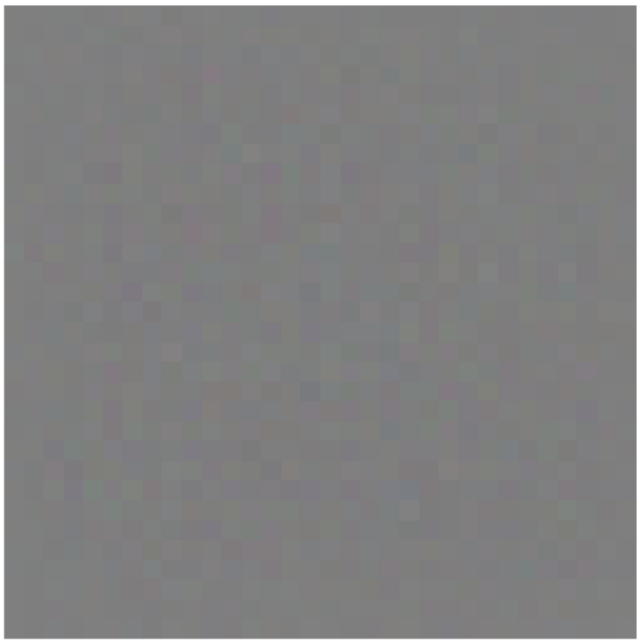
Listing 1: Architektura dyskryminatora

Następnie przystępuję do implementacji modelu *generatora*. W opisie zadania nie były w nim wprost uwzględnione warstwy *batch normalization*, ale dodaję je - pomiędzy parami warstw konwolucji transponowanej oraz funkcji aktywacji **leaky relu**.

```
generator = models.Sequential()
generator.add(layers.Input(shape=(64,)))
generator.add(layers.Dense(units=4*4*64))
generator.add(layers.Reshape(target_shape=(4, 4, 64)))
generator.add(layers.Conv2DTranspose(
    filters=64, kernel_size=(4, 4), strides=(2, 2), padding="same"
))

generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU(negative_slope=0.2))
generator.add(layers.Conv2DTranspose(
    filters=128, kernel_size=(4, 4), strides=(2, 2), padding="same"
))

generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU(negative_slope=0.2))
generator.add(layers.Conv2DTranspose(
    filters=256, kernel_size=(4, 4), strides=(2, 2), padding="same"
))

generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU(negative_slope=0.2))
generator.add(layers.Conv2D(
    filters=3, kernel_size=(5, 5), padding="same", activation="tanh"
))
```

Listing 2: Architektura generatora

Poniżej wyświetlam wygenerowany przez generator obraz (dla losowego szumu o rozkładzie normalnym, średniej w 0 i odchyleniu standardowym równym 1). Jak widać, nie obrazuje on (na razie) niczego konkretnego.



Teraz przygotowuję zbiór obrazów do treningu. Wykorzystuję udostępniony w ramach zadania mniejszy zbiór **prawdziwych** zdjęć ciast marchewkowych (*crawled_cakes*). W zbiorze treningowym znajduje się dokładnie 975 obrazów przedstawiających ciasta marchewkowe.

```
# wczytanie obrazów z ciastami marchewkowymi
dataset = image_dataset_from_directory(
    directory="./crawled_cakes/",
    labels=None,
    label_mode=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(32, 32),
    shuffle=True,
    seed=42,
    interpolation="bilinear",
    verbose=False
)

# normalizacja obrazów do zakresu wartości [-1, 1]
normalizer = layers.Rescaling(1. / 127.5, offset=-1)
dataset = dataset.map(lambda image: (normalizer(image)))
```

Listing 3: Przygotowanie zbioru treningowego

Jak widać, użyty tutaj rozmiar batcha wynosi 32 - tak zatem będzie także podczas treningu. Wyświetlam parę przykładowych obrazków, żeby upewnić się, że wszystko gra:

(a) (b) (c)

Rysunek 1: Pierwsze 3 obrazki dataset'u - odpowiednio (a), (b) i (c)

Następnie na przykładzie małego modelu sprawdzę możliwości oferowane przez framework w zakresie "ręcznego" treningu sieci.

```
# mały model na "rozruch"
simple_model = models.Sequential()
simple_model.add(layers.Input(shape=(3,)))
simple_model.add(layers.Dense(units=3))
simple_model.add(layers.Dense(units=3))

# losowanie wejścia do modelu
input = np.random.uniform(low=20., high=64., size=(8, 3)).astype('float32')

# używamy optymalizatora SGD
optimizer = optimizers.SGD(learning_rate=1e-3)

# używana funkcja kosztu - jak zaproponowano w poleceniu
loss_fn = lambda output: tf.math.reduce_mean(output) - 42
```

```
# "ręczny" trening przez 5 epok
for epoch in range(5):
    print(f"Zaczynam epokę treningu nr {epoch + 1}:")

    with GradientTape() as tape:
        output = simple_model(input, training=True)
        loss = loss_fn(output)

    # odzyskanie gradientów
    gradients = tape.gradient(loss, simple_model.trainable_weights)

    # aplikowanie spadku po gradiencie
    optimizer.apply(gradients, simple_model.trainable_weights)

    # wypisanie wartości funkcji kosztu dla epoki
    print(f"Wartość (pseudo)funkcji kosztu dla epoki nr {epoch+1}: {float(loss):.4f}")
```

Listing 4: Mały model "na rozruch" i jego trening

Wypisywane na wyjściu wartości istotnie maleją (co prawda są ujemne, w "normalnej" funkcji kosztu tak się nie dzieje, ale to jest przecież *pseudofunkcja* kosztu).

Nadchodzi pora na właściwy trening dyskryminatora oraz generatora. Nie pokażę całego kodu, gdyż jest go całkiem dużo, za to umieszczę wybrane, istotne fragmenty. Jedynym ustalanym parametrem treningu jest liczba epok, bo rozmiar batcha jest uwzględniony w datasecie. W optymalizatorach Adam ustalam parametr `beta_1=0.5`, gdyż znalazłem informację, że taka praktyka jest stosowana w GAN-ach.

```
# ustalenie parametrów treningu
EPOCHS = 2500

# używana funkcja kosztu
loss_fn = losses.binary_crossentropy

# optymalizatory - osobne dla dyskryminatora i generatora
discriminator_optimizer = optimizers.Adam(learning_rate=1e-5, beta_1=0.5)
generator_optimizer = optimizers.Adam(learning_rate=1e-5, beta_1=0.5)
```

Listing 5: Parametry trenigu wraz z używaną funkcja kosztu i optymalizatorami

Przedstawiam wspólnie części pętli treningowej dla dyskryminatora i generatora.

```
for step, real_images in enumerate(dataset):
    # ustalanie rozmiaru batcha (ostatni może być mniejszy)
    batch_size = real_images.shape[0]

    # trening dyskryminatora
    generated_input = generate_random_input(batch_size=batch_size)
    generated_images = generator(generated_input, training=False)

    # połączone batche oraz etykiety
    input = tf.concat([real_images, generated_images], axis=0)
    labels = tf.convert_to_tensor([1.] * batch_size + [0.] * batch_size, dtype=tf.float32)

    # zmiana wymiaru potrzebna do funkcji kosztu
    labels = tf.expand_dims(labels, axis=-1)

    # zaciemnienie wektora etykiet
    labels += np.random.uniform(low=-0.05, high=0.05, size=(2*batch_size, 1))
        .astype('float32')
```

```
    # podanie batcha na wejście dyskryminatora
    with GradientTape() as tape:
        output = discriminator(input, training=True)
        loss_discriminator = loss_fn(labels, output)

    # policzenie pochodnych po wagach dyskryminatora
    gradients = tape.gradient(loss_discriminator, discriminator.trainable_weights)

    # aktualizacja wag dyskryminatora
    discriminator_optimizer.apply(gradients, discriminator.trainable_weights)

    # trening generatora
    with GradientTape() as tape:
        # nowy batch wygenerowanych obrazów
        generated_input = generate_random_input(batch_size=batch_size)
        generated_images = generator(generated_input, training=True)

        # wektor oczekiwanych etykiet "na odwrót"
        labels = tf.convert_to_tensor([1.] * batch_size, dtype=tf.float32)
        labels = tf.expand_dims(labels, axis=-1)

        # podanie wygenerowanych obrazów na wejście dyskryminatora
        output = discriminator(generated_images, training=False)
        loss_generator = loss_fn(labels, output)

    # policzenie pochodnych po wagach generatora
    gradients = tape.gradient(loss_generator, generator.trainable_weights)

    # aktualizacja wag generatora
    generator_optimizer.apply(gradients, generator.trainable_weights)
```

Listing 6: Pojedyncza epoka treningu dyskryminatora i generatora

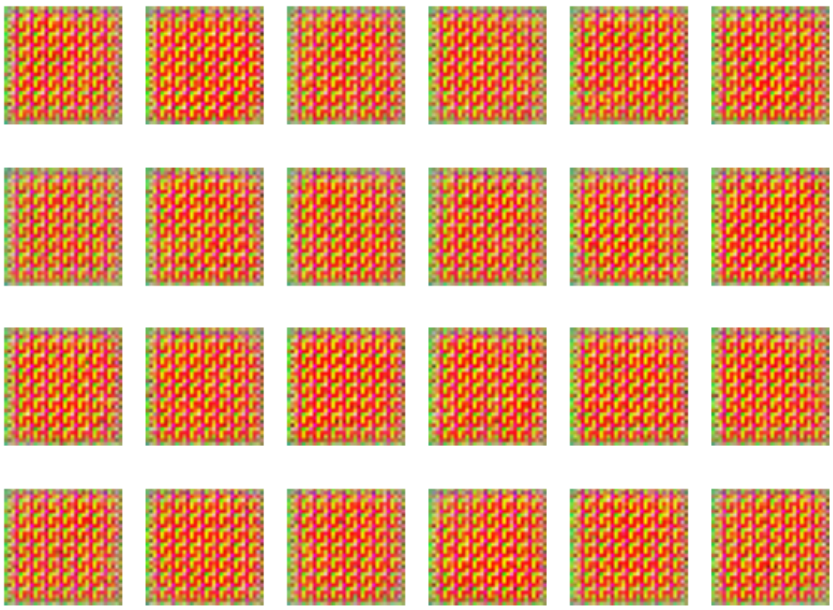
W moim przypadku taka jedna epoka treningu trwa około $20s$. Biorąc pod uwagę przyjęte 2500 epok, raczej jednym ciągiem całości treningu nie uda się wykonać. Dlatego na potrzeby zapisywania stanu modeli, optymalizatorów i epoki treningu korzystam z tzw. *checkpoint'ów*. W kerasie są to obiekty, które automatycznie zapisują stan w formacie rozumianym przez bibliotekę i umożliwiającym późniejsze odtworzenie. Ponadto, jeden checkpoint może obejmować model oraz powiązany z nim optymalizator, zatem w moim przypadku wystarczą dwa takie obiekty.
Przyjmuję zapis w pierwszych 10 epokach (aby na początku kontrolować rozwój wydarzeń, czy któryś z modeli nie przeważa nad drugim), następnie co 10 epok i finalnie co 100 epok począwszy od setnej epoki.
Wtedy również wypisuję na wyjście uśrednione wartości funkcji kosztu dla obu modeli oraz zapisuję wygenerowane przez generator (dla ustalonego przed treningiem, losowego szumu na wejściu) obrazy. Realizuję to następująco:

```
# wektory "testowe" na potrzeby badania postępów generatora
test_vectors = generate_random_input(batch_size=24)

# funkcja służąca do zapisywania generowanych obrazów w określonych epokach
def generate_and_save_images(generator, epoch):
    generated_images = generator(test_vectors)
    _, ax = plt.subplots(4, 6)

    for i, image in enumerate(generated_images):
        # kroki analogiczne jak przy wyświetlaniu obrazków
        image_to_save = (image + 1.) / 2.
        image_to_save = np.clip(image_to_save, 0., 1.)
        ax[i//6][i%6].imshow(image_to_save)
        ax[i//6][i%6].axis('off')
```

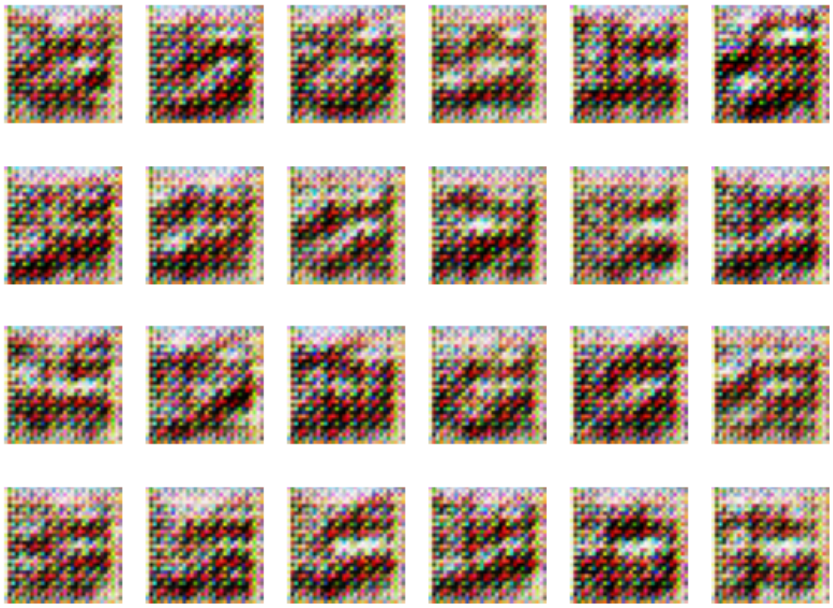
```
plt.savefig(f"./saved_epochs/epoch_{epoch}.png")
plt.close()
```

Listing 7: Funkcja służąca do cyklicznego zapisu generowanych obrazów

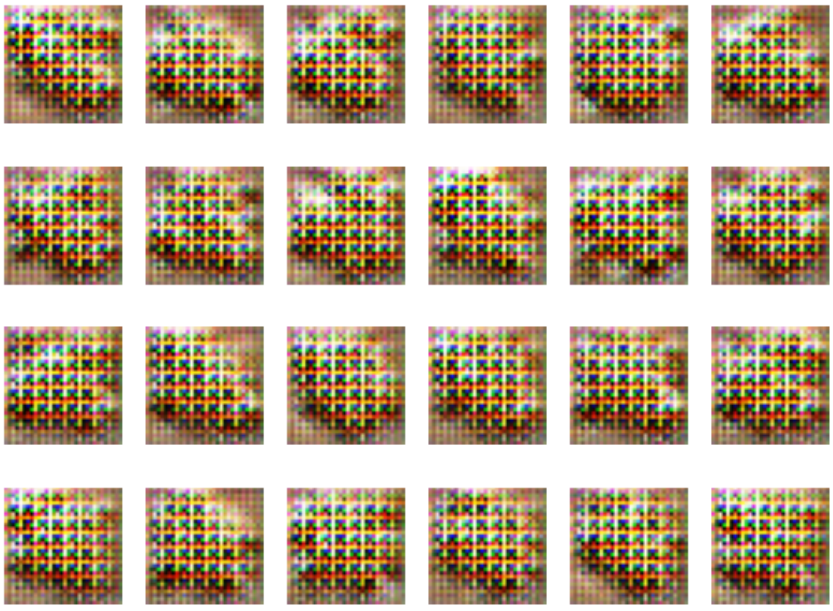
Po dłuuugim treningu uznaję, że najlepiej (choć efekt nie powala na kolana) wyglądają obrazy wygenerowane w epoce dość wczesnej, bo 600-tnej. Rezultaty oceniam negatywnie, ale cieszę się, że w niektórych przypadkach obrazy choć trochę mogą przypominać pożądane ciasta marchewkowe. Jednakże stwierdzenie, że są one podobne i mogłyby kogoś zmylić jest mocno na wyrost - obrazy są zdecydowanie niskiej jakości. Poniżej przedstawiam, jak wyglądają wygenerowane obrazy w wybranych epokach.



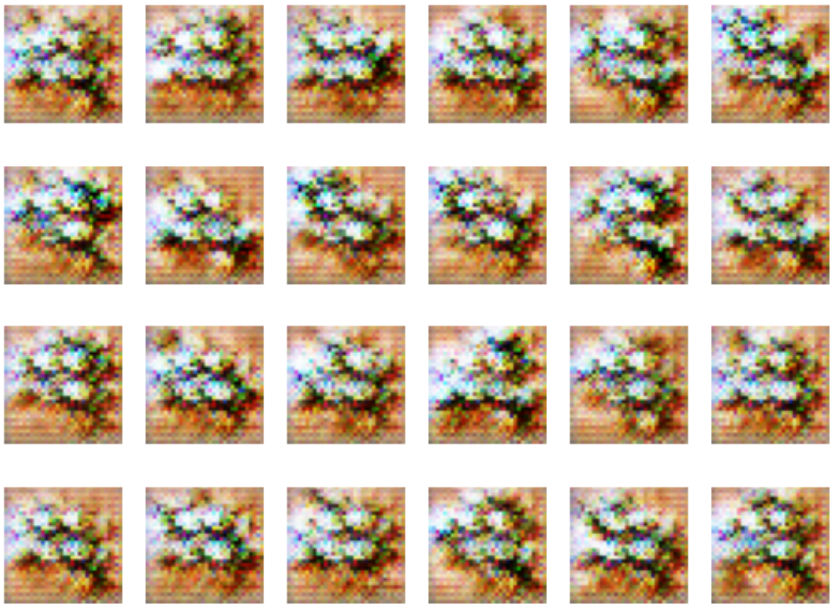
Rysunek 2: Obrazy generowane po 10 epokach - na razie nic szczególnego



Rysunek 3: Obrazy generowane po 600 epokach - uznałem, że wyglądają najlepiej



Rysunek 4: Obrazy generowane po 1500 epokach - jeden z rozpatrywanych kandydatów



Rysunek 5: Obrazy generowane po 1600 epokach - wyglądają bardziej jak kwiaty!

Rysunek 6: Obrazy generowane po 2500 epokach - koniec, niestety nie wygląda to sensownie

Teraz zaprezentuję działanie inwersji generatora, czyli optymalizacji wektora wejściowego, a nie parametrów sieci. Do tego celu używam stanu generatora z epoki nr 600 - wczytuję to z odpowiedniego checkpointu:

```
# najpierw tworzę ścieżkę do pliku
generator_path = os.path.join(
    generator_checkpoint_manager.directory,
    "ckpt-24"
)

# wczytanie stanu
generator_status = generator_checkpoint.restore(generator_path)

# sprawdzenie stanu generatora
if generator_status.expect_partial():
    print(f"Pomyślnie wczytano stan generatora z {generator_path}")
else:
    print(f"Nie udało się wczytać checkpointa generatora z {generator_path}")
```

Listing 8: Wczytanie stanu generatora z 600. epoki

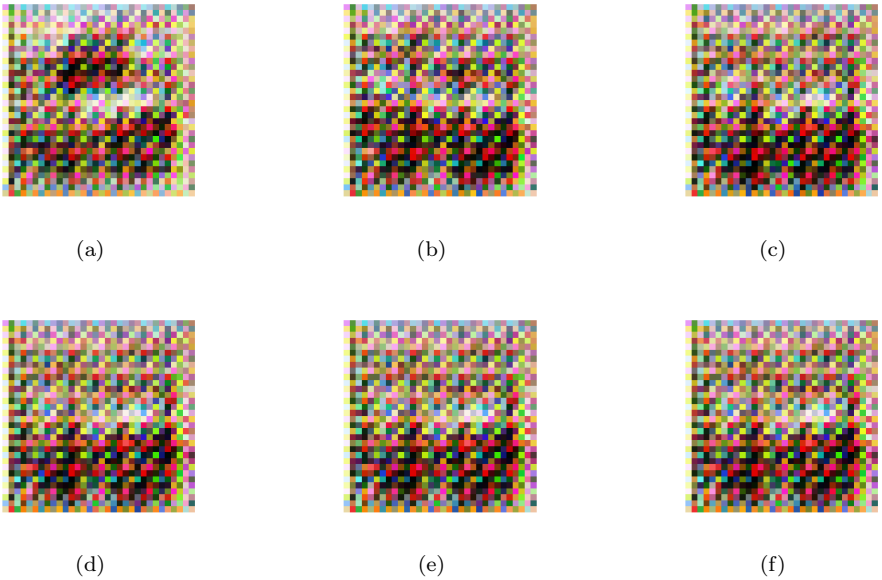
Jako że proces będzie się powtarzał kilkukrotnie, opakowuję kod w funkcję. Zgodnie z poleceniem zadania używam jako funkcji kosztu błędu średniokwadratowego **MSE**. Z kolei optymalizator to **SGD** z parametrami `momentum=0.9` oraz `learning_rate=0.01`. Przyjmuję `ITERS=100` iteracji procesu optymalizacji.

```
def optimize_input_vector_to_image(input_vector, image, optimizer, save_directory):
    # miejsce zapisywania obrazów "pośrednich"
    os.makedirs(save_directory, exist_ok=True)

    # wymagane, aby móc liczyć gradienty po wektorze wejściowym
    input_vector_variable = tf.Variable(input_vector, trainable=True)

    # pętla optymalizacji
    for i in range(ITERS):
        with GradientTape() as tape:
            output = generator(input_vector_variable, training=False)
            loss = loss_fn(image, output)

        # obliczenie gradientów po wektorze wejściowym
        gradients = tape.gradient(loss, input_vector_variable)

        # aplikowanie spadku po gradiencie
        optimizer.apply([gradients], [input_vector_variable])

        if (i+1) in ITERS_SAVE:
            # wypisanie wartości funkcji kosztu
            print(f"Wartość funkcji kosztu dla iteracji nr {i+1}: {float(tf.reduce_sum(loss))}")

            # zapisanie obrazu
            generated_image = generator(input_vector_variable)[0]
            image_to_save = (generated_image + 1.) / 2.
            image_to_save = np.clip(image_to_save, 0., 1.)

            plt.imshow(image_to_save)
            plt.axis('off')
            plt.savefig(f"{save_directory}/iter_{i+1}.png")
            plt.close()

    return input_vector_variable.numpy()
```

Listing 9: Funkcja dokonująca optymalizacji wektora wejściowego

Najpierw uruchamiam tę procedurę dla poniższego obrazu ze zbioru treningowego.

W tym i w następnych przykładach stosuję zapis wygenerowanych obrazów w iteracjach: 1, 10, 25, 50, 75 i 100. Ewolucja wygenerowanych obrazów (a) - (f), odpowiadająca tej kolejności wygląda w tym przypadku tak:



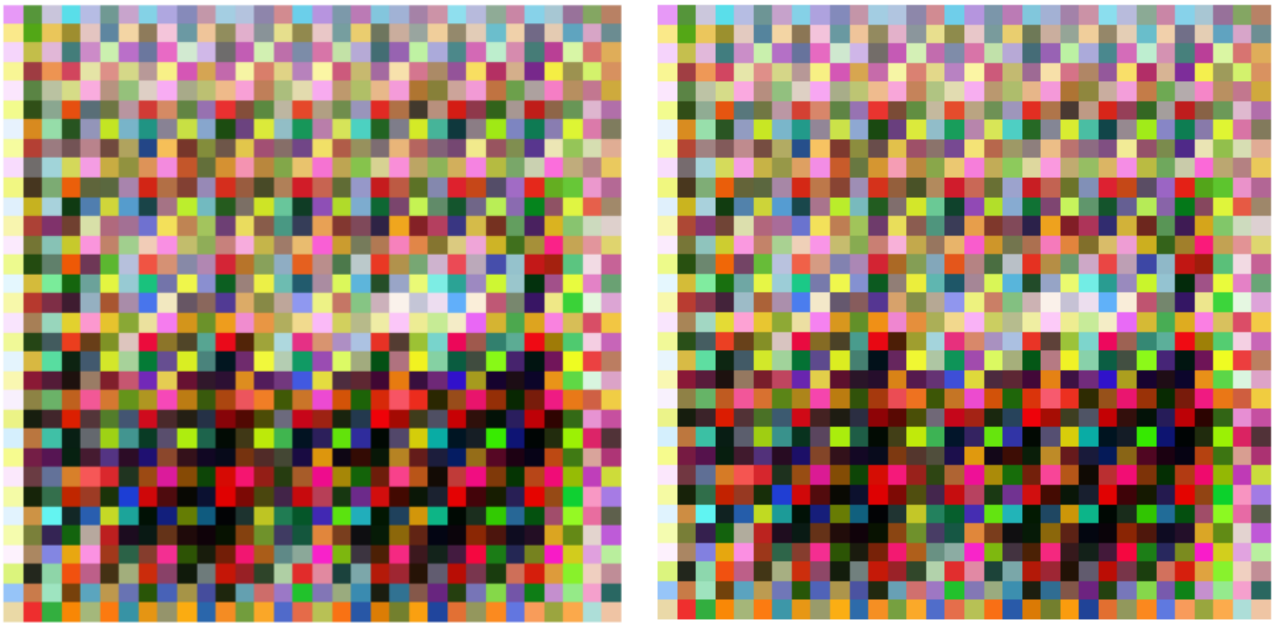
(a) (b) (c)

(d) (e) (f)

Rysunek 7: Ewolucja wygenerowanych obrazów

Jak widać, mocno rezonują artefakty ("szkocka krata"), które były widoczne przy okazji prezentacji obrazów z różnych epok. Występuje niewielki ruch "w stronę" obrazu, który chciałem odtworzyć, ale jakość finalnego obrazka nie jest wysoka.

Po zmianie kilku wartości w wektorze wyjściowym (który jest zwracany przez pokazaną wyżej funkcję) różnica jest raczej niezauważalna:

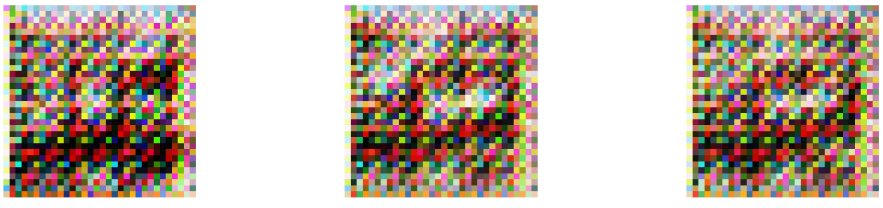


(a) (b)

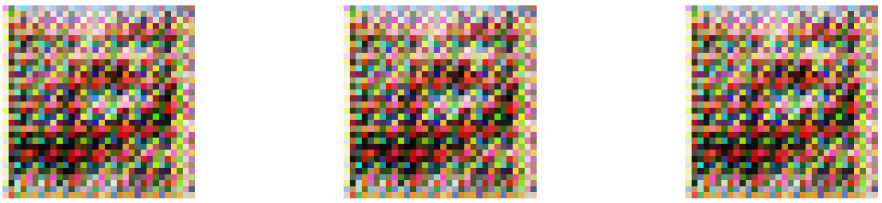
Rysunek 8: Porównanie obrazu z ostatniej iteracji (a) i obrazu wygenerowanego ze zmienionego wektora (b)

Następnie powtarzam eksperyment, ale dla obrazu, który jest zupełnie niepowiązany z wytrenowaną klasą. Mianowicie jest to obraz przedstawiający samochód. Po przekształceniach (takich samych, jak przy tworzeniu dataset'u z obrazów treningowych) prezentuje się następująco:

Dla tego obrazu ewolucja generowanych obrazów (a) - (f) nie odbiega od poprzedniego przypadku.



(a) (b) (c)

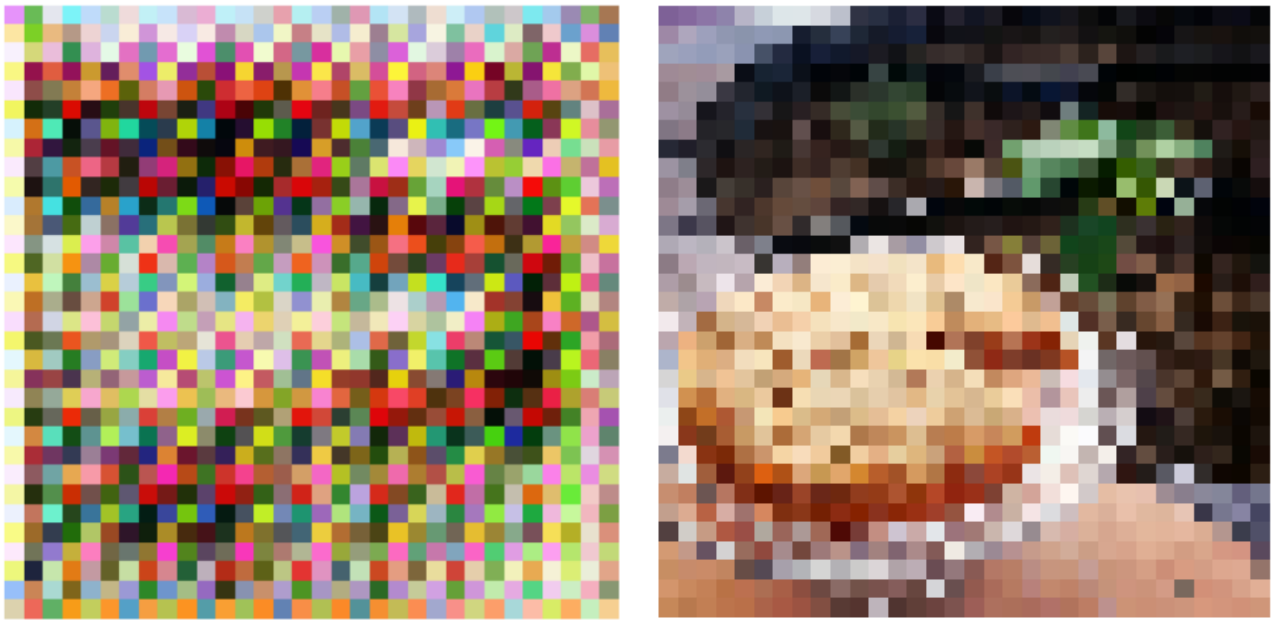


(d) (e) (f)

Rysunek 9: Ewolucja wygenerowanych obrazów

Jeżeli się dokładnie przyjrzymy, widać różnicę - zarówno między obrazkami w tej ewolucji, jak i pomiędzy ewolucjami dla obu dotychczas badanych obrazów. Na ostatnim obrazku znajduje się kształt nieco przypominający samochód.

Przechodzę do ostatniej części zadania - powtarzam eksperyment dla innego obrazu ze zbioru treningowego, następnie tworzę interpolację między wektorem otrzymanym dla pierwszego obrazu a wektorem otrzymanym z tego drugiego obrazu.

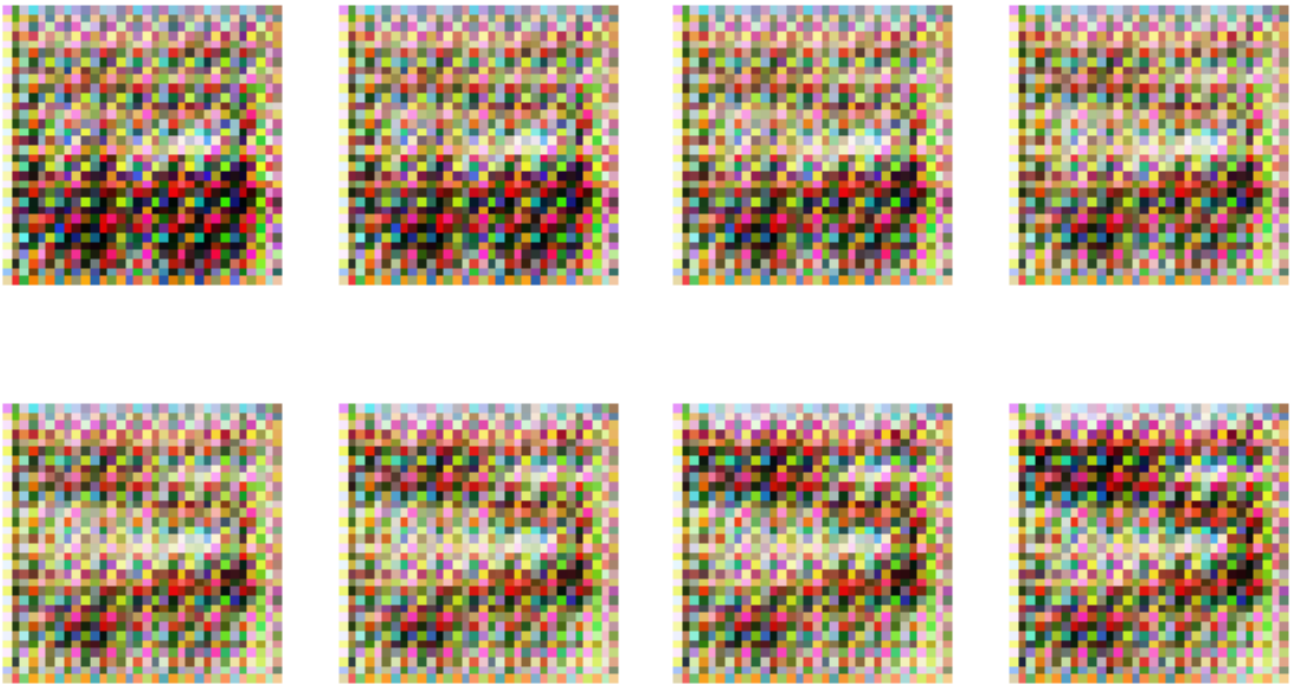


(a) (b)

Rysunek 10: Wygenerowany obraz z ostatniej iteracji optymalizacji (a) na podstawie obrazu (b)

Nie ukrywam - trzeba dużo dobrej woli, żeby zobaczyć podobieństwo między tymi obrazami, ale minimalne takie podobieństwo jest.

Włącznie z wektorami "brzegowymi" generuję obrazy dla 8 wektorów (zatem pomiędzy nimi dokonuję interpolacji 6 wektorów). Ciąg "pośrednich" obrazów przedstawiam poniżej. Uwzględniam także obrazy dla obu uzyskanych wektorów. Jako pierwszy jest obraz dla pierwszego wektora, ostatni to obraz dla drugiego wektora, a pomiędzy nimi stopniowo zmniejsza się udział wektora pierwszego, za to zwiększa drugiego.



Widać delikatne przejście od jednego obrazu do drugiego. Nie wybija się ono szczególnie, wynika to z podobieństwa tych obrazów (a to z kolei jest skutkiem nienajlepszego generatora).

Wnioski i wrażenia

Niewątpliwie to zadanie przysporzyło mi wiele trudności. Przede wszystkim początkowo zabrakło mi trochę wiary i cierpliwości, bo patrząc na początkowe rezultaty (100-200 epok), kiedy generowane obrazy przedstawiały kolorową kratkę (i to praktycznie wszystkie taką samą, mimo że losowane były przecież różne wektory) oraz na wartości liczbowe uśrednionej funkcji kosztu dla sieci, które potrafiły się różnić diametralnie między sobą, podejmowałem decyzję o zmianie parametrów i ponownym uruchomieniu treningu od nowa - a samo doliczenie do przykładowo 200. epoki na moim sprzęcie trwało około godziny. Zabrakło mi podejścia *trust the process* - aż w końcu, nieco zrezygnowany, zostawiłem początkowe wartości parametrów, pozwoliłem sieciom uczyć się do skutku... i coś wyszło. Niekoniecznie idealne, ale wyszło.
Przechodząc do kwestii jakości wyników: na pewno nie jest ona wysoka. Nie powiedziałbym, że jestem zawiedziony, ale liczyłem na coś trochę lepszego. Zyskując nieco doświadczenia podczas realizacji tego laboratorium wiem, że trening takiego GAN-a w "domowych warunkach" nie jest łatwy, było to zresztą wspomniane w poleceniu. Dlatego cieszę się, że udało się uzyskać coś lepszego niż losowy szum i nawet jeżeli efekt nie jest spektakularny, to zrealizowałem wszystkie zadania z podstawowej części laboratorium.
Być może, gdybym pozwolił na dłuższy trening (500, może 1000 epok więcej), to udałoby sie uzyskać lepszy generator. Nie udało się tego sprawdzić z powodu wspomnianych problemów i faktu, że jednak trochę za późno zabrałem się za realizację tego zadania. Jako że nie chcę przekroczyć terminu, te rozważania zapewne pozostaną w sferze gdybania.