

Metody rozpoznawania obrazów

Zadanie 5 – raport

Mateusz Kocot

6 marca 2022

Spis treści

1	Model dyskryminatora	1
2	Model generatora	2
3	Zbiór danych	2
4	Trening	3
4.1	Rozruch	3
4.2	Trening dyskryminatora – implementacja	5
4.3	Trening generatora – implementacja	5
4.4	Trening całości – implementacja	6
4.5	Trening	6
5	Generator z upsamplingiem	9
5.1	Implementacja	9
5.2	Trening	9
6	Eksperymenty	12
6.1	Kod	12
6.2	Pierwszy model generatora	13
6.3	Drugi model generatora (z upsamplingiem)	16
7	Podsumowanie	19

1 Model dyskryminatora

Najpierw utworzyłem funkcję tworzącą blok konwolucyjny:

```
def convolution_block(n_filters):  
    return [tf_layers.Conv2D(n_filters, (4, 4), strides=(2, 2),  
                           padding='same'),
```

```
    tf_layers.BatchNorm(),
    tf_layers.LeakyReLU(alpha=0.2)]
```

Cały model dyskryminatora tworzony jest przez kolejną funkcję:

```
def make_discriminator_model():
    return tf.keras.Sequential([
        tf.keras.Input(shape=(64, 64, 3)),
        *convolution_block(64),
        *convolution_block(128),
        *convolution_block(128),
        tf_layers.Flatten(),
        tf_layers.Dropout(0.2),
        tf_layers.Dense(1, activation='sigmoid')
    ])
```

2 Model generatora

Podobnie jak poprzednio, zaimplementowałem funkcję pomocniczą tworzącą blok konwolucji transponowanej:

```
def transpose_convolution_block(n_filters):
    return [tf_layers.Conv2DTranspose(n_filters, (4, 4), strides=(2, 2),
                                    padding='same'),
            tf_layers.LeakyReLU(alpha=0.2)]
```

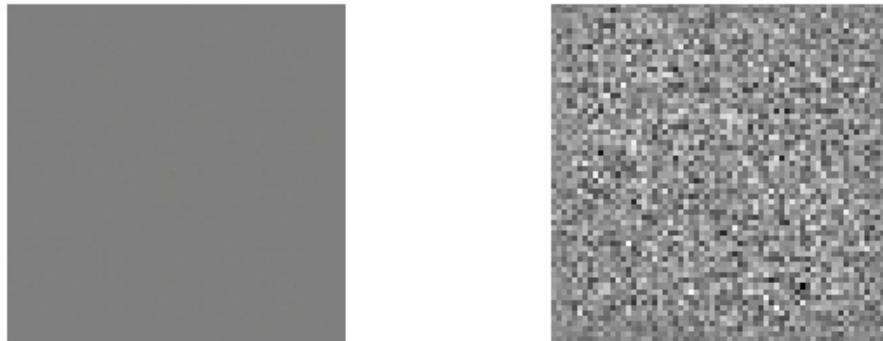
Cały model generatora tworzony jest w następujący sposób:

```
def make_generator_model():
    return tf.keras.Sequential([
        tf.keras.Input(shape=(128,)),
        tf_layers.Dense(8192, # 8 * 8 * 128 = 8192
                      ),
        tf_layers.Reshape((8, 8, 128)),
        *transpose_convolution_block(128),
        *transpose_convolution_block(256),
        *transpose_convolution_block(512),
        tf_layers.Conv2D(3, (5, 5), strides=(1, 1), padding='same',
                        activation='sigmoid')
    ])
```

Co ciekawe, po podaniu na wejście losowego szumu, utworzone obrazy mają w całości ten sam kolor (rys. 1). Poszczególne kanały są już jednak różne (rys. 1).

3 Zbiór danych

Wybrałem zbiór danych składający się ze zdjęć spaghetti bolognese. Korzystając z Duck-DuckGo udało się uzyskać całkiem sporo – 345 – sensownych zdjęć. Wadą zbioru jest dość



Rys. 1: Obraz utworzony przed podanie na wejście niewytrenowanego generatora losowego szumu. Po lewej stronie cały obraz RGB, po prawej – tylko pierwszy kanał

duża szczegółowość zdjęć – makaron może być poplątany na nieskończonym wiele sposobów. Wszystkie zdjęcia są natomiast do siebie podobne – typowe zdjęcie to zmiękowane kolory czerwony i żółty znajdujące się na talerzu. Ewentualnie, na środku może być kilka zielonych listków. Przykłady przedstawiłem na rys.2.



Rys. 2: Przykładowe obrazy ze zbioru danych

Obrazy w każdej iteracji treningu będą nieco odmienne dzięki użyciu *data augmentation*:

```
data_augmentation = tf.keras.Sequential([
    tf_layers.RandomFlip('horizontal'),
    tf_layers.RandomZoom(0.1),
    tf_layers.RandomRotation(0.1)
])
```

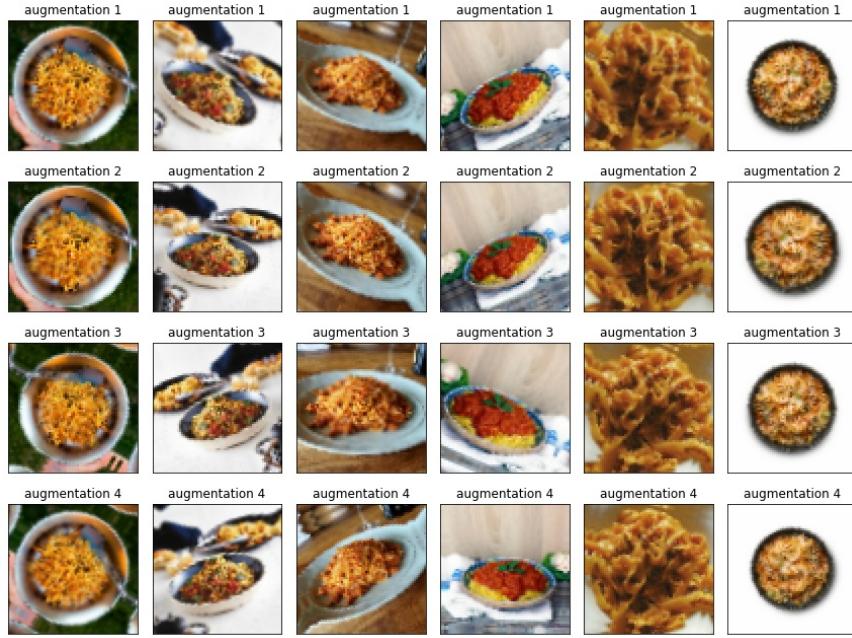
Kilka przykładów różnych wariantów danych zdjęć pokazałem na rys. 3.

4 Trening

4.1 Rozruch

Przygotowałem model składający się z dwóch warstw:

```
test_model = tf.keras.Sequential(layers =[
```



Rys. 3: Przykład *data augmentation* – różne warianty przykładowych zdjęć

```
        tf.keras.Input(shape=(3,)),
        tf_layers.Dense(5),
        tf_layers.Dense(3)
    ])
```

Jako funkcję straty przyjąłem sumę różnic sum wyjścia i liczby 10:

```
def test_loss(output):
    return tf.math.reduce_sum(tf.abs(
        tf.math.reduce_sum(output, axis=1) - tf.constant(10.)))
```

Na wejście sieci podałem batch składający się z 5 losowych wektorów:

```
tf.random.uniform((5, 3))
```

Uzyskałem wartość funkcji straty: 42.41. Wartości pochodnych cząstkowych liczone są w osobnej funkcji:

```
def test_gradient(model, input_):
    weights = model.layers[0].trainable_variables
    with tf.GradientTape() as tape:
        output = test_loss(model(input_))
        gradient = tape.gradient(output, weights)
    return gradient
```

Ostatecznie wykonałem 20 iteracji korygowania wag z użyciem optymalizatora SGD:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
for i in range(1, 21):
```

```

gradient = test_gradient(test_model, test_input)
optimizer.apply_gradients(
    zip(gradient, test_model.layers[0].trainable_weights))

```

Wartość funkcji loss spadała z każdą iteracją. Po 10 iteracjach wynosiła 28.21, a po 20 – już tylko 14.01.

4.2 Trening dyskryminatora – implementacja

Zaimplementowałem funkcję przeprowadzającą jedną turę treningu dyskryminatora. Na wejście, oprócz modeli dyskryminatora i generatora, dostaje ona batch prawdziwych obrazów, rozmiar batcha, funkcję straty oraz optymalizator. Na wyjściu zwracana jest wartość funkcji straty.

```

@tf.function
def discriminator_round(discriminator, generator, real_images,
                        batch_size, loss_fun, optimizer):
    noise = tf.random.normal((batch_size, 128))

    with tf.GradientTape() as tape:
        generated_images = generator(noise, training=False)
        X = tf.concat([real_images, generated_images], axis=0)
        y = tf.concat([tf.ones(batch_size), tf.zeros(batch_size)], axis=0)
        y += tf.random.uniform((2 * batch_size,), -0.05, 0.05)

        loss = loss_fun(y, discriminator(X, training=True))
        gradient = tape.gradient(loss, discriminator.trainable_variables)

    optimizer.apply_gradients(
        zip(gradient, discriminator.trainable_variables))
    return loss

```

Dzięki dodaniu dekoratora `@tf.function`, funkcja została skompilowana przed wykonaniem, dzięki czemu proces treningu przebiegał nieco szybciej.

4.3 Trening generatora – implementacja

Funkcja odpowiedzialna za pojedynczą turę treningu generatora jest podobna do poprzedniej. Tym razem, nie ma jednak potrzeby przyjmowania na wejście prawdziwych obrazów.

```

@tf.function
def generator_round(discriminator, generator, batch_size,
                     loss_fun, optimizer):
    noise = tf.random.normal((batch_size, 128))

    with tf.GradientTape() as tape:
        X = generator(noise, training=True)

```

```

y = tf.ones(batch_size)

loss = loss_fun(y, discriminator(X, training=False)) # ...
gradient = tape.gradient(loss, generator.trainable_variables)

optimizer.apply_gradients(zip(gradient, generator.trainable_variables))
return loss

```

4.4 Trening całości – implementacja

Zaimplementowałem dwie funkcje pomocnicze. `gan_round` wywołuje funkcję trenującą dyskryminator, a następnie generator. `gan_epoch` odpowiada za całą epokę i zwraca średnie wartości funkcji straty. Warto zaznaczyć, że w procesie treningu wykorzystałem osobne optymalizatory *Adam* dla obu sieci. Dzięki odpowiedniemu operowaniu parametrami `learning_rate`, wagi obu sieci pozostawały we względnej równowadze.

Ostatecznie, zaimplementowałem funkcję `train_gan`, która trenuje sieci oraz zbiera statystyki. Z ważniejszych rzeczy, definiowany jest batch losowych wektorów:

```
noise_vectors = np.random.normal(loc=0.0, scale=1.0, size=(24, 128))
```

Przy zapisywaniu modelu, zapisywane są także obrazy. Wykorzystywana jest funkcja pomocnicza `plot_sample_images`. `model_name` to argument funkcji `train_gan`; `epoch` aktualizowany jest na bieżąco.

```

sample_images = generator(noise_vectors, training=False)
plot_sample_images(sample_images,
                   f'plots/{model_name}_training/sample_images_{epoch}')

```

4.5 Trening

Po wielu eksperymentach, dobrałem optymalizatory tak, by wartości funkcji straty obu sieci pozostawały w przybliżonej równowadze:

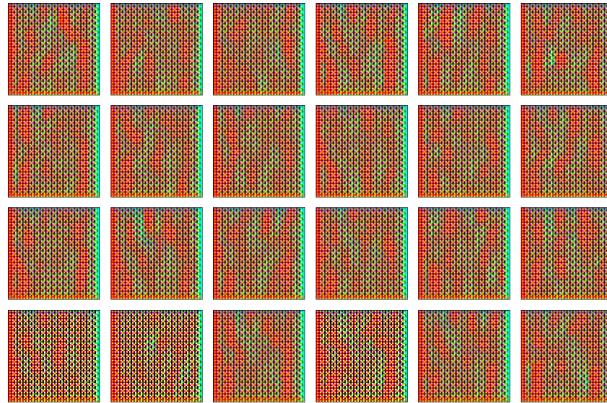
```

d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.000001)
g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.00001)

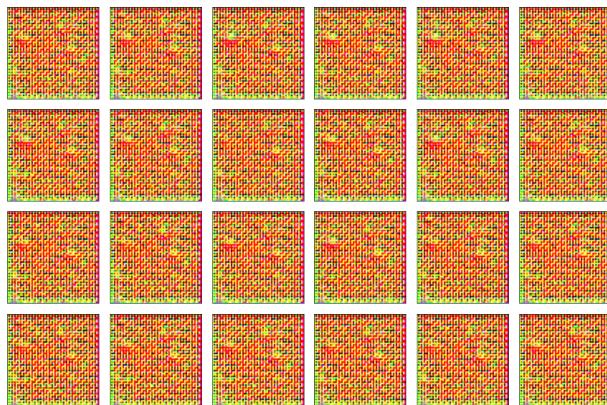
```

Trening trwał w sumie 5700 epok. Kilka razy był przerwany, a potem model był wczytywany z jednego z zapisanych punktów kontrolnych. Poza początkowymi większymi niż normalnie wahaniem funkcji straty wynikającymi z użycia świeżych optymalizatorów, nie wpłynęło to na przebieg treningu.

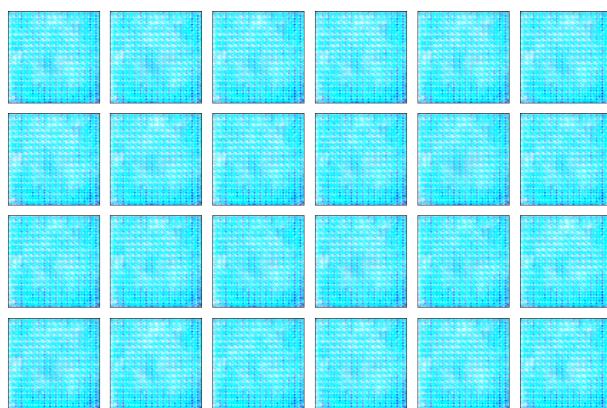
Niestety, od samego początku wszystkie generowane obrazy były bardzo do siebie zbliżone. Model nie uczył się zwracać najmniejszej uwagi na dane wejściowe (o ile były to dane zgodne z rozkładem normalnym). Zamiast z tego, z iteracji na iterację dość gwałtownie zmieniał generowane obrazy. Ewolucję generacji przedstawiłem na rysunkach od 4 do 9.



Rys. 4: Obrazy generowane przez pierwszy generator po 10 epokach treningu. Pierwsze iteracje były bardzo burzliwe. Po 50 epokach dominował kolor żółty; po 70 – kolor niebieski.



Rys. 5: Obrazy generowane przez pierwszy generator po 100 epokach treningu. Pojawiają się zarysy coraz bardziej złożonych kształtów.



Rys. 6: Obrazy generowane przez pierwszy generator po 700 epokach treningu. Od 250 iteracji przeważnie można już wyodrębnić obiekt na środku od tła. Po 700 epokach pojawia się chyba najbardziej skrajny wyjątek.



Rys. 7: Obrazy generowane przez pierwszy generator po 1000 epokach treningu. Coraz częściej zaczyna pojawiać się żółto-czerwona papka z domieszką białego albo czarnego (talerze) i/lub zielonego (liście).



Rys. 8: Obrazy generowane przez pierwszy generator po 3000 epokach treningu. Po 1000 epokach trening przestał być już tak spektakularny jak wcześniej, ale z biegiem czasu coraz częściej pojawiają się próby imitacji makaronu.



Rys. 9: Obrazy generowane przez pierwszy generator po 5400 epokach treningu. Jeden z najładniejszych obrazów – mimo ciemnego rogu. Ten model wykorzystam w późniejszych eksperymentach.

5 Generator z upsamplingiem

5.1 Implementacja

Nowy generator korzysta z dwóch funkcji pomocniczych:

```
def convolution_block(n_filters):
    return [tf_layers.Conv2D(n_filters, (3, 3), padding='same'),
            tf_layers.BatchNormalization(),
            tf_layers.LeakyReLU(alpha=0.2)]

def upsampling_block(n_filters):
    return [tf_layers.UpSampling2D((2, 2)),
            *convolution_block(n_filters),
            *convolution_block(n_filters)]
```

Funkcja tworząca model nowego generatora:

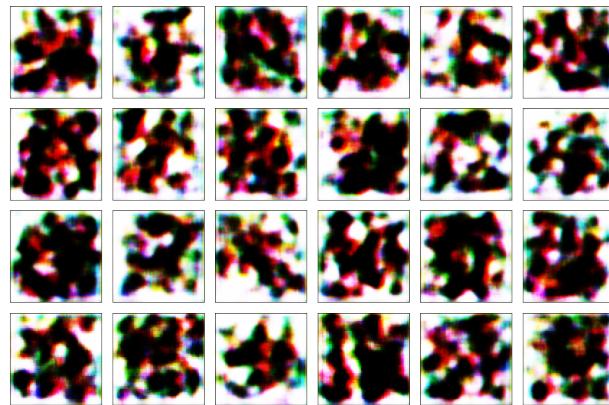
```
def make_generator_with_upsampling():
    return tf.keras.Sequential([
        tf.keras.Input(shape=(128,)),
        tf_layers.Dense(8192),
        tf_layers.Reshape((8, 8, 128)),
        *upsampling_block(64),
        *upsampling_block(128),
        *upsampling_block(256),
        tf_layers.Conv2D(3, (5, 5), padding='same', activation='sigmoid')
    ])
```

5.2 Trening

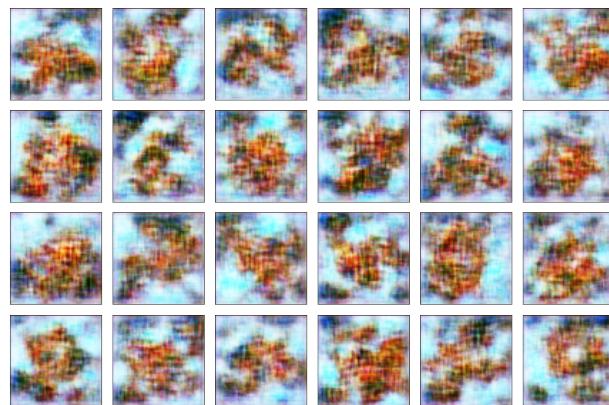
Tym razem użyłem nieco inne optymalizatory:

```
d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.000001)
g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.000008)
```

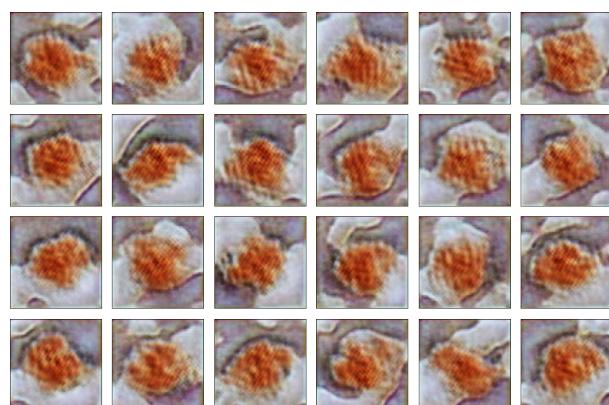
Trening trwał w sumie 4600 epok; przebiegał zupełnie inaczej niż w poprzednim przypadku, lecz na koniec znowu model zaczął generować te same obrazy – niezależnie od wejścia. Wcześniej, mimo że generowane obrazki były różne, to wciąż przypominały raczej różne wersje jednego obrazu, a model przecież powinien próbować odwzorować każdy obraz ze zbioru. Obrazy z różnych etapów treningu przedstawiłem na rysunkach od 10 do 15.



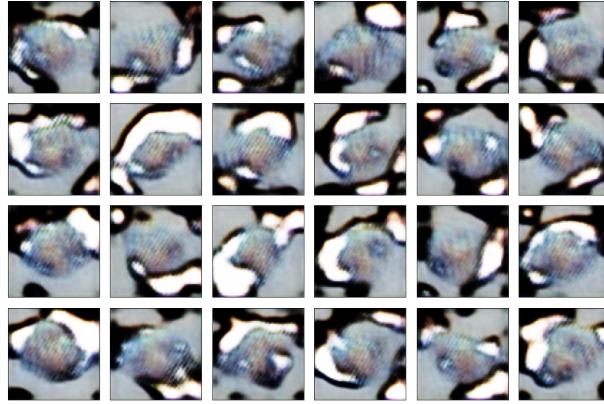
Rys. 10: Obrazy generowane przez generator z upsamplingiem po 10 epokach treningu. Od początku tworzone obrazy są nieco różne.



Rys. 11: Obrazy generowane przez generator z upsamplingiem po 100 epokach treningu. Pojawiają się już poprawne dominujące barwy.



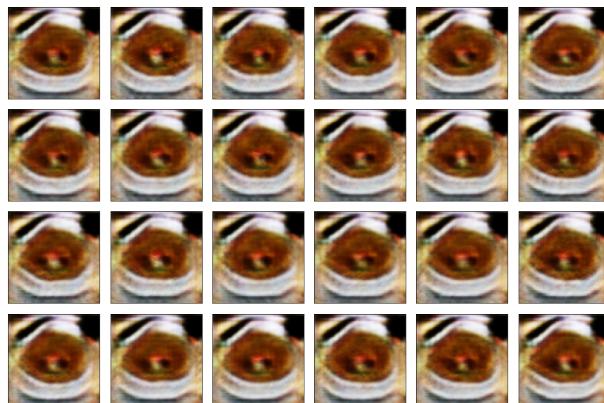
Rys. 12: Obrazy generowane przez generator z upsamplingiem po 950 epokach treningu. Centralne części obrazków zaczynają przypominać spaghetti bolognese.



Rys. 13: Obrazy generowane przez generator z upsamplingiem po 1000 epokach treningu. Trening nie zawsze jest stabilny. W ciągu 50 epok generator nauczył się generować całkowicie inne obrazy od wcześniejszych.



Rys. 14: Obrazy generowane przez generator z upsamplingiem po 2500 epokach treningu. Jeden z ładniejszych przypadków. Ten model wykorzystam w późniejszych eksperymentach.



Rys. 15: Obrazy generowane przez generator z upsamplingiem po 4200 epokach treningu. Od 3500 epoki generator znowu zaczął generować niemal identyczne obrazy. Najbardziej skrajny przypadek widoczny był 4200 epokach – generator niemal odtworzył jeden z prawdziwych obrazów.

6 Eksperymenty

6.1 Kod

Kod, który przedstawię poniżej tworzy losowy wektor szumu a następnie go „optymalizuje”, tzn. próbuje znaleźć taki wejściowy wektor by wygenerować dany obraz `selected_image`. Po kilku eksperymentach, jako optymalizator wybrałem Adam z `learning_rate` równym aż 1.0.

```
noise = tf.Variable(tf.random.normal((1, 128)))

loss_fun = tf.keras.losses.MeanSquaredError()
optimizer = tf.keras.optimizers.Adam(learning_rate=1.)

for i in range(40):
    with tf.GradientTape() as tape:
        loss = loss_fun(selected_image, generator(noise, training=False))
        gradient = tape.gradient(loss, [noise])

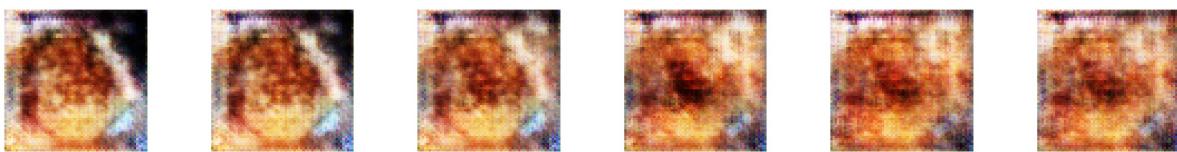
    optimizer.apply_gradients(zip(gradient, [noise]))
```

6.2 Pierwszy model generatora

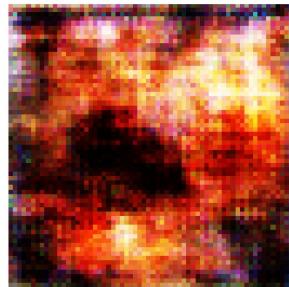
Tutaj nie spodziewam się spektakularnych rezultatów – w końcu generator generuje te same obrazy niezależnie od wejścia. Wyniki tej sekcji przedstawiłem poniżej.



Rys. 16: Obraz, który będę próbował odwzorować w pierwszym kroku



Rys. 17: Model 1: ewolucja generowanego obrazu po 0, 1, 4, 10, 20 i 40 epokach optymalizacji wejściowego szumu. Optymalizator zrobił co mógł – udało się usunąć czarną plamę oraz dodać niezbyt udaną imitację sera na środku. Model niestety nie pozwolił na więcej.



Rys. 18: Model 1: generowany po 40 epokach obraz ze zmodyfikowanym szumem. Co piąta wartość została przemnożona przez 5. Obraz jest bardziej „skrajną” wersją poprzedniego.



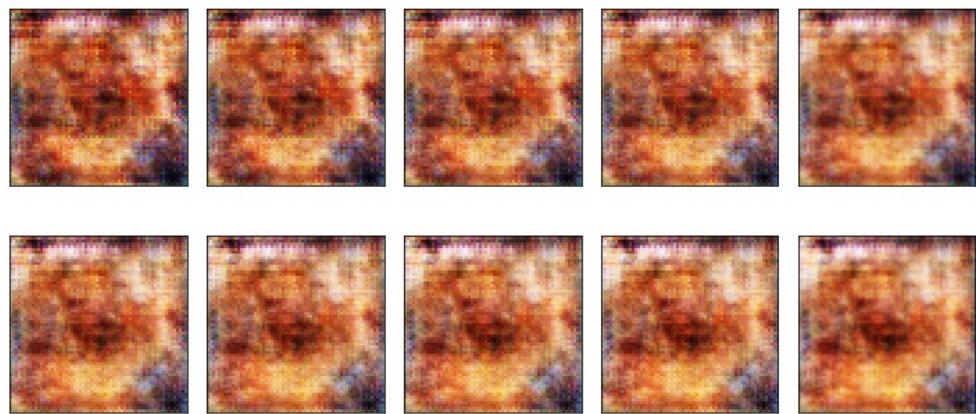
Rys. 19: Obraz niepowiązany z wytrenowaną klasą – Airbus Beluga wykorzystany przy jednym z poprzednich zadań



Rys. 20: Model 1 – obraz niezwiązany z wytrenowaną klasą: ewolucja generowanego obrazu po 0, 1, 4, 10, 20 i 40 epokach optymalizacji wejściowego szumu. Efekt nie jest spektakularny, ale na trzech ostatnich obrazkach można zauważać poziome przyciemnienie w przybliżeniu mające podobne kontury do samolotu.



Rys. 21: Drugi wykorzystany obraz ze zbioru treningowego



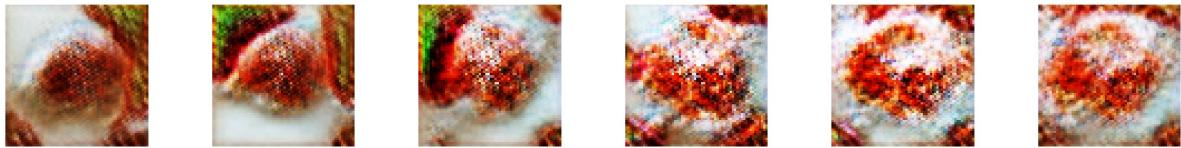
Rys. 22: Model 1: lewy górny róg – obraz wygenerowany przy użyciu wektora odpowiadającego drugiemu obrazkowi, prawy dolny róg – obraz wygenerowany przy użyciu wektora odpowiadającego pierwszemu obrazkowi. Pozostałe obrazki to odpowiednie średnie ważone (od lewej do prawej, wierszami). Nie ma widocznej różnicy

6.3 Drugi model generatora (z upsamplingiem)

Tutaj rezultaty powinny być nieco lepsze.



Rys. 23: Obraz, który będę próbował odwzorować w pierwszym kroku



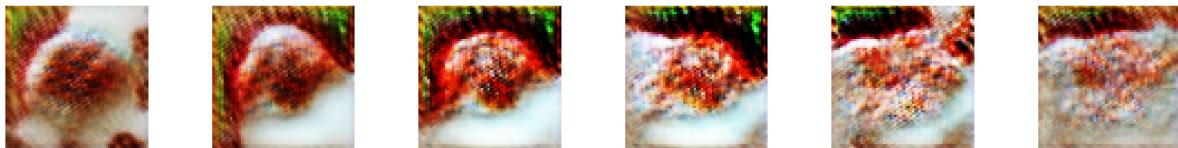
Rys. 24: Model 2: ewolucja generowanego obrazu po 0, 1, 4, 10, 20 i 40 epokach optymalizacji wejściowego szumu. Ostateczny obraz jest niezłym odwzorowaniem bazowego. Paleta kolorów jest nieco inna, ale widać spore podobieństwo



Rys. 25: Model 2: generowany po 40 epokach obraz ze zmodyfikowanym szumem. Co piąta wartość została przemnożona przez 5. Niektóre części obrazu zostały całkowicie odmienione – to na nie modyfikacja szumu miała największy wpływ



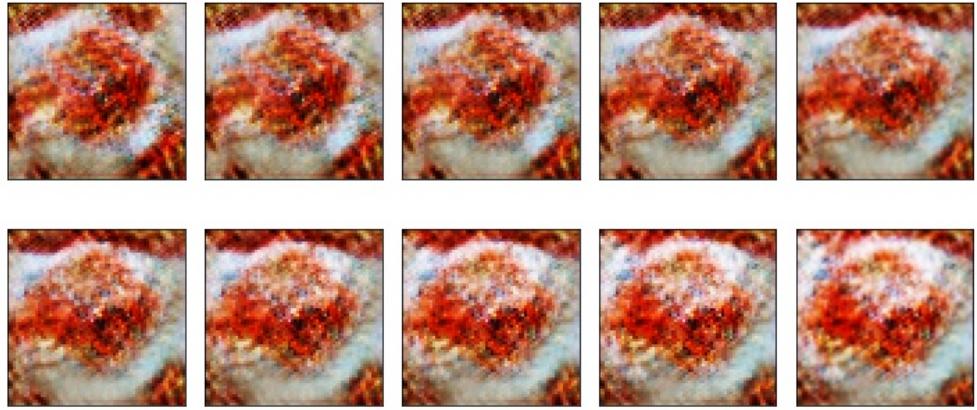
Rys. 26: Obraz niepowiązany z wytrenowaną klasą – Airbus Beluga wykorzystany przy jednym z poprzednich zadań



Rys. 27: Model 1 – obraz niezwiązany z wytrenowaną klasą: ewolucja generowanego obrazu po 0, 1, 4, 10, 20 i 40 epokach optymalizacji wejściowego szumu. Dół bazowego obrazu jest jasny (chmury, samolot), a góra ciemna, niebieska (niebo) – generowany obraz całkiem nieźle to odwzorował. Generator nie został wyposażony w możliwość generowania koloru niebieskiego, ale jakoś i tak sobie poradził.



Rys. 28: Drugi wykorzystany obraz ze zbioru treningowego



Rys. 29: Model 1: lewy górny róg – obraz wygenerowany przy użyciu wektora odpowiadającego drugiemu obrazkowi, prawy dolny róg – obraz wygenerowany przy użyciu wektora odpowiadającego pierwszemu obrazkowi. Pozostałe obrazki to odpowiednie średnie ważone (od lewej do prawej, wierszami). Różnica jest subtelna, ale da się ją zinterpretować. Pierwszy generowany obrazek ma widoczne ciemne fragmenty – prawdopodobnie są to listki. Widać, jak powoli zamieniają się w bardziej dominujący w pierwszym obrazie bazowym ser.

7 Podsumowanie

Zrobienie całego zadania, włącznie z napisaniem raportu, zajęło mi bardzo dużo czasu. Najwięcej czasu spędziłem na dostosowywaniu wag, tak żeby wartości funkcji straty generatora i dyskryminatora były we względnej równowadze. Sporo czasu próbowałem też znaleźć rozwiązanie problemu generowania tych samych obrazów, tj. *mode collapse* – problem ten wydaje się być dość powszechny, lecz nie znalazłem rozwiązania. Najbardziej denerwującą częścią okazała się ostatnia część zadania. Wygenerowałem tyle wykresów, że orientowanie się w tym wszystkim było dość wymagające. Najbardziej satysfakcjonuje były momenty, gdy generatory zaczynały w końcu generować mające cokolwiek wspólnego z rzeczywistością obrazy.

W przypadku poprzednich zadań często nasuwał się wniosek, że stosunkowo łatwe modele sieci neuronowych są w stanie wykonać złożone zadania. W tym przypadku, tak łatwe sieci niestety nie wystarczyły.