

Generowanie obrazów w stylu Claude'a Moneta

Aleksandra Buchowicz, Bartosz Jabłoński, Tomasz Markowicz, Filip Pazio

Spis treści

1	Temat projektu	3
2	Zbiór danych	3
3	Wprowadzenie do sieci GAN	5
4	Implementacja	7
5	Wyniki	10
6	Podsumowanie i wnioski	12
7	Źródła	13

1 Temat projektu

Tematem projektu było generowanie obrazów na wzór twórczości Claude'a Moneta przy pomocy sieci DCGAN (ang. *Deep Convolutional Generative Adversarial Network*). Dodatkowo zbadaliśmy, jak dobór rozkładu, z którego generowany jest szum i następnie wykorzystywany w modelu do tworzenia nowych obrazów wpłynie na końcowy wynik. Sieć zaimplementowano w języku Python, korzystając z notatnika Jupyter Notebook i dodatkowych modułów, m.in. *PyTorch*.

2 Zbiór danych

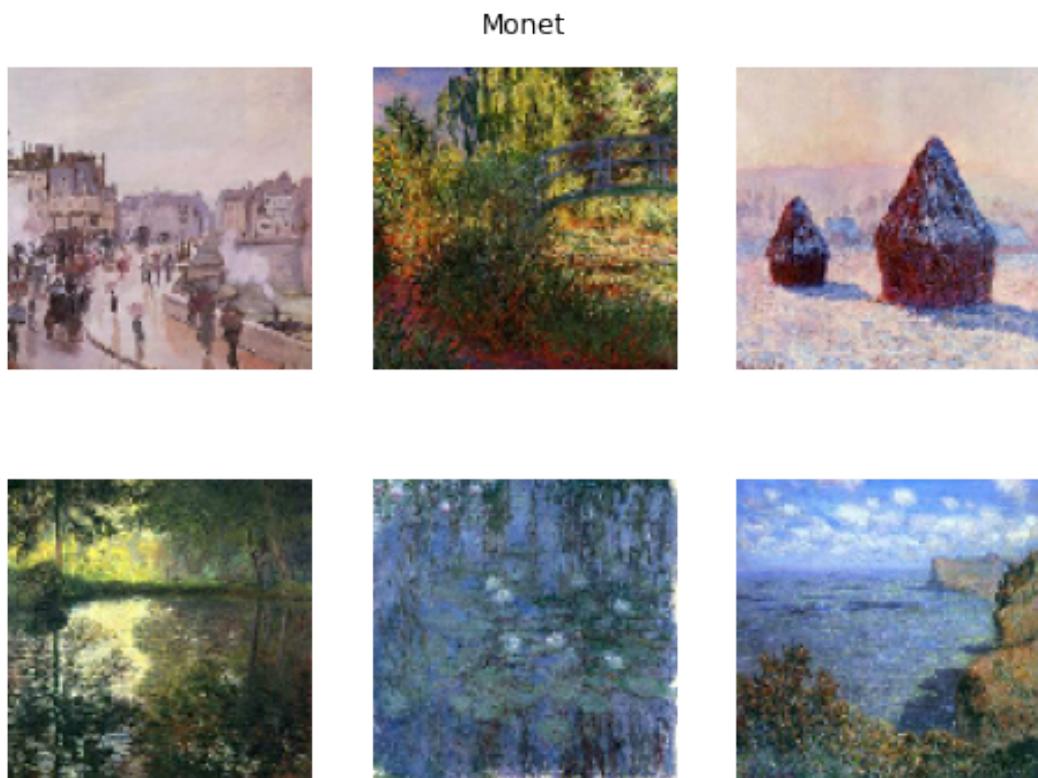
Wykorzystany zbiór danych pochodzi z serwisu Kaggle: <https://www.kaggle.com/competitions/gan-getting-started>. Zbiór składa się z następujących folderów:

- **monet.jpg** - 300 obrazów Claude'a Moneta o wymiarach 256x256 w formacie JPEG
- **monet_tfrec** - 300 obrazów Claude'a Moneta o wymiarach 256x256 w formacie TFRecord
- **photo.jpg** - 7028 fotografii o wymiarach 256x256 w formacie JPEG
- **photo_tfrec** - 7028 fotografii o wymiarach 256x256 w formacie TFRecord

Tematem naszego projektu było wykorzystanie sieci DCGAN do tworzenia nowych obrazów z szumu. W związku z tym korzystaliśmy jedynie z plików z folderu **monet.jpg**.

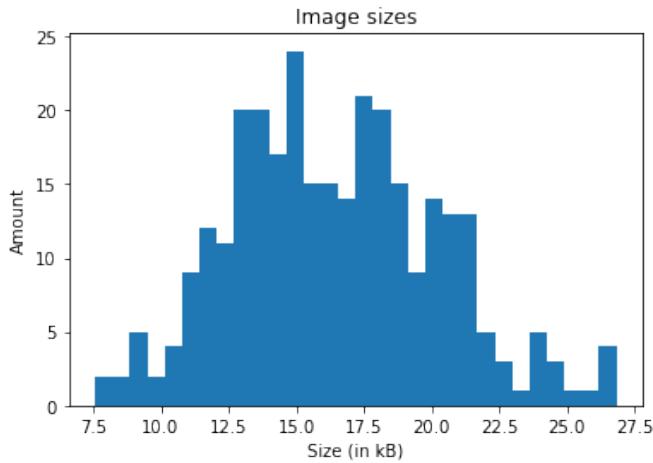
Eksploracja danych

Poniżej zaprezentowano przykłady obrazów zawartych w zbiorze danych:



Rysunek 1: Przykładowe obrazy z folderu **monet.jpg**

Sprawdziliśmy również jakich rozmiarów są pliki ze zbioru treningowego. Wyniki przedstawiliśmy w formie histogramu:



Rysunek 2: Rozmiary plików zawartych w folderze `monet-jpg`

W dalszej części projektu będziemy operować nie na samych obrazach, ale na tensorach reprezentujących obrazy. Postanowiliśmy więc od razu przekształcić obrazy na tensory i wyliczyliśmy ich wartości minimalne, maksymalne i średnie, jak również wariancję i odchylenie standardowe. Nasze wyniki przekształciliśmy z powrotem na obrazy.



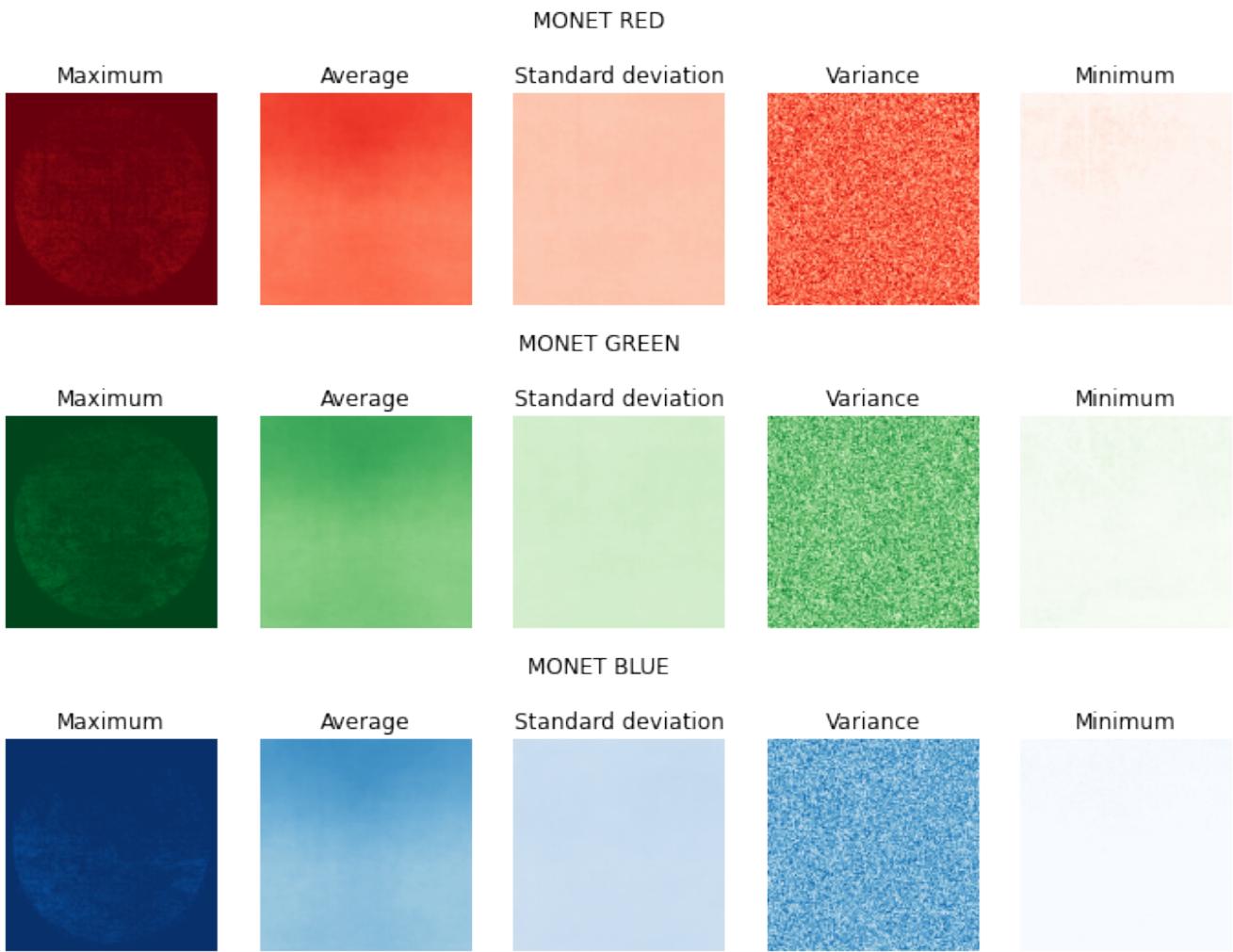
Rysunek 3: Różne statystyki wartości reprezentujących obrazy z folderu `monet-jpg`

W przypadku wartości średnich można zauważyc zarys horyzontu. Innym ciekawym wynikiem jest powstanie "kółka" przy wyliczaniu wartości maksymalnej. Przyczyną tego zjawiska jest występowanie w zbiorze danych obrazów malowanych na kolistym płótnie.



Rysunek 4: Przykłady obrazów Claude'a Moneta na kolistym płótnie

Sprawdziliśmy również, jak wcześniejsze statystyki będą się zmieniać dla poszczególnych kolorów.



Rysunek 5: Różne statystyki wartości reprezentujących obrazy z folder `monet.jpg`

3 Wprowadzenie do sieci GAN

GANy (ang. *Generative Adversarial Networks*) stanowią strukturę do tworzenia modeli głębokiego uczenia maszynowego wyłapujących rozkład danych treningowych, co umożliwia następnie generowanie nowych danych z tego samego rozkładu. Sieci GAN zostały wynalezione w 2014 roku przez Iana Goodfellowa ([3]). Charakteryzuje je wykorzystanie dwóch oddzielnych modeli - generatora i dyskryminatora.

Zadaniem generatora jest tworzenie "fałszywych" obrazów, które wyglądają jak obrazy treningowe. Zadaniem dyskryminatora jest rozpoznawanie, czy dany obraz jest prawdziwym obrazem z danych treningowych czy fałszywym, wygenerowanym przez generator. Podczas treningu generator stara się "oszukać" dyskryminator, tworząc obrazy coraz bardziej zbliżone do obrazów treningowych. Z kolei dyskryminator stara się lepiej klasyfikować obrazy jako fałszywe lub nie. Celem treningu jest osiągnięcie stanu, gdy generator produkuje obrazy nieroróżnicialne od obrazów treningowych.

DCGAN jest rozwinięciem struktury GAN opisanej powyżej, przy czym w szczególności wykorzystuje ona warstwy konwolucyjne (splotowe) zarówno w generatorze, jak i dyskryminatorze. Zostały one po raz pierwszy opisane przez Radforda i współautorów w 2016 roku ([4]).

Generator

Niech z będzie wektorem pochodząącym z pewnego rozkładu (np. jednostajnego). $G(z)$ reprezentuje funkcję generatora, która przekształca z na element przestrzeni danych treningowych. Zadaniem G jest wyestymowanie rozkładu, z którego pochodzą dane treningowe (ozn. p_{data}), w celu generowanie fałszywych próbek z tego wyestymowanego rozkładu (ozn. p_g).

W praktyce przekształcenie jest osiągane poprzez serię dwu-wymiarowych, transponowanych konwolucji. Po każdej konwolucji, z wyjątkiem ostatniej, wykonywana jest częściowa normalizacja, a następnie funkcję prostującą (ang. *rectifier* lub *ReLU*)

Częściowa normalizacja jest określona wzorem

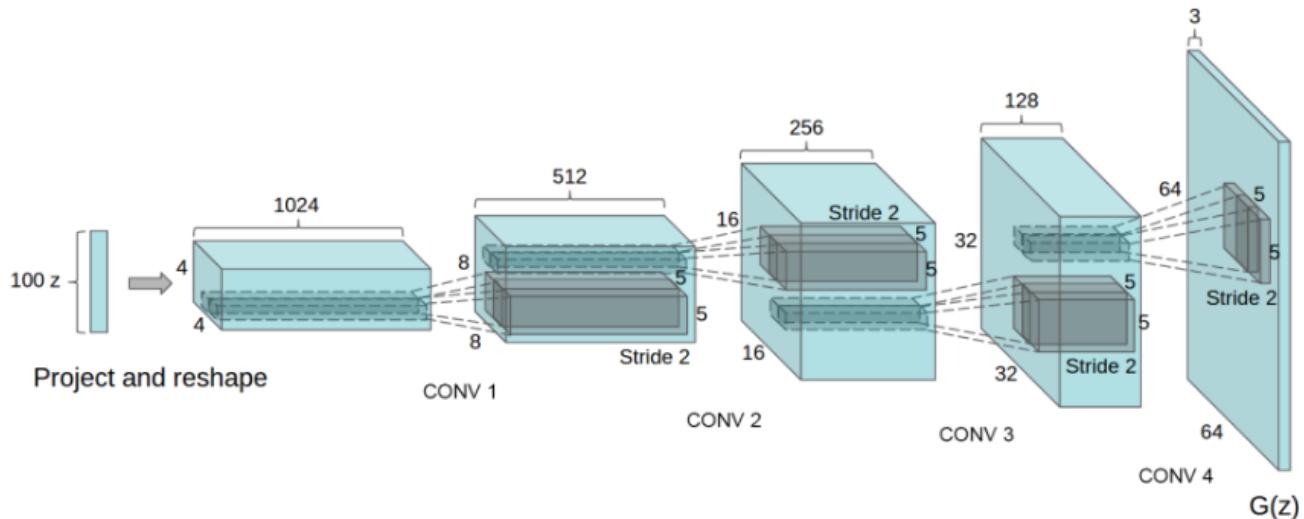
$$\frac{x - E[x]}{\sqrt{Var[x] - \epsilon}} * \gamma + \beta$$

Średnia i odchylenie standardowe są wyliczane według wymiaru względem małych części, zaś γ i β są wyuczone wektory parametryczne o rozmiarze C (gdzie C jest rozmiarem wejścia). Domyślnie elementy γ są ustawione na 1, z kolei elementy β są ustawione na 0. Odchylenie standardowe jest liczone poprzez estymator stronniczy (*biased estimator*).

Funkcja prostująca jest funkcją określoną po kolejnych argumentach

$$ReLU(x) = \max(0, x)$$

Wyjście generatora przechodzi przez funkcję *tanh* celem otrzymania wartości z przedziału [-1,1]. Warto zwrócić uwagę na pojawianie się normalizacji po kolejnych konwolucjach, gdyż ułatwiają one modelowi przepływ gradientów podczas treningu. Docelowo będziemy chcieli otrzymać tensor $3 \times 64 \times 64$, która odpowiada za 3 warstwy RGB o rozmiarach identycznych do rozmiaru danych wejściowych (obrazów). Ogólną strukturę generatora obrazuje rysunek podany poniżej (Rysunek 6).



Rysunek 6: Struktura warstw konwolucyjnych generatora

Dyskryminator

Niech x będzie tensorem o wymiarach $3 \times 64 \times 64$ reprezentującym obraz. $D(x)$ to sieć dyskryminatora, która zwraca prawdopodobieństwo (wartość skalarną), że x pochodzi z danych treningowych, a nie z generatora. Intuicyjnie, wartość $D(x)$ powinna być duża, jeśli x pochodzi z danych treningowych, oraz mała, jeżeli x pochodzi z generatora. $D(x)$ może być rozumiany jako tradycyjny klasyfikator binarny. Dyskryminator, podobnie jak generator, składa się z warstw konwolucyjnych, ale tym razem bez transpozycji. Po każdej konwolucji, oprócz wejściowej i wyjściowej, wykonywana jest częściowa normalizacja, która była opisana w sekcji opisującej generator, a następnie nakładana jest nieszczelna wersja funkcji prostującej (*ang. LeakyReLU - Leaky Rectified Linear Unit*) określona wzorem

$$\text{LeakyReLU}(x) = \max(0, x) + s * \min(0, x)$$

gdzie s jest parametrem kontrolującym nachylenie części ujemnej argumentów. Po ostatniej konwolucji wykonywane jest nałożenie funkcji *Sigmoid*.

Funkcja straty

Wobec tego $D(G(Z))$ jest prawdopodobieństwem, że dane zwarcane przez generator G pochodzą z danych treningowych. Zatem dyskryminator D będzie starał się maksymalizować prawdopodobieństwo, że dobrze zaklasyfikowało obraz jako prawdziwy lub fałszywy ($\log(D(x))$), a generator G stara się zminimalizować prawdopodobieństwo, że D zaklasyfikuje obrazy wygenerowane przez G jako obrazy fałszywe ($\log(1 - D(G(z)))$). W związku z tym, funkcja straty GAN przyjmuje postać

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(D(x))] + \mathbb{E}_{x \sim p_g(z)}[\log(1 - D(G(z)))]$$

W teorii rozwiązanie powyższego zadania minmax zostało znalezione, jeśli $p_g = p_{\text{data}}$ i dyskryminator zaklasyfikuje obrazy jako prawdziwe lub fałszywe z prawdopodobieństwem 50%. W praktyce modele nie zawsze trenują do osiągnięcia tego celu, a teoria zbieżności dla sieci GAN jest ciągle rozwijana.

4 Implementacja

Zbudowany przez nasz model dla wygody został zawarty w jednej klasie `DCGAN`, która wykorzystuje dwie inne klasy `Discriminator` i `Generator` tworzące odpowiednio dyskryminator i generator. W dalszej części tej sekcji omówimy najistotniejsze metody klasy `DCGAN`: `weights_init`, `train_generator` oraz `train_discriminator`.

Przed treningiem obrazy są załadowywane, przekształcane na tensory i normalizowane do przedziału $[-1, 1]$, zgodnie z dobrymi praktykami opisanymi w [5].

Wymagane biblioteki

- `os` - dla określania ścieżki do katalogu z obrazami (danymi) niezależnie od urządzenia
- `NumPy` - obsługa wielowymiarowych tabel i macierzy, analiza numeryczna
- `PyTorch` - framework open-source wspomagający projektowanie modeli uczenia maszynowego
- `PIL` (Python Imaging Library) – rozszerzenie dla Pythona, które dodaje obsługę grafiki np. otwieranie, modyfikowanie, zapisywanie plików graficznych.
- `pickle` - serializacja i deserializacja struktur obiektów
- `matplotlib` - tworzenie wykresów i animacji

Wagi

Zgodnie z artykułem [4] wszystkie wagi użyte w optimizerach modelu z rozkładu normalnego o wartości oczekiwanej 0 i odchyleniu standardowym 0.02. Metoda `weights_init` przyjmuje na wejściu zainicjowany model i reiniicjalizuje wszystkie warstwy konwolucyjne, konwolucyjno-transponowane oraz warstwy normalizacji partii, żeby spełniały ten warunek. Metoda ta jest wywoływana na modelu generatora i dyskryminatora od razu po inicjalizacji pełnego modelu DCGAN.

Algorithm 1 Inicjalizacja wag

- 1: zainicjuj `weights_init(model)`
 - 2: `classname` \leftarrow nazwa modelu
 - 3: jeśli w `classname` nie występuje człon "Conv": ustaw wagi z rozkładu $N(0, 0.02)$
 - 4: wpp. jeśli w `classname` nie występuje człon "BatchNorm": ustaw wagi z rozkładu $N(0, 0.02)$ i obciążenie 0
-

Funkcje straty i optimizery

Do obliczania straty dyskryminatora i generatora wykorzystamy funkcję Binary Cross Entropy Loss (**BCELoss**) z pakietu PyTorch, która jest definiowana w następujący sposób:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, \quad l_n = -[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

Zauważmy, że funkcja ta pozwala na wyliczenia zarówno straty generatora $\log(1 - D(G(z)))$ jak i dyskryminatora $\log(D(x))$ przez podanie odpowiedniego y , czyli etykiet treningowych. Według konwencji ustalonej w [3] etykiety prawdziwe powinny być zdefiniowane jako 1 a fałszywe jako 0. Jednak podążając za dobrymi praktykami opisanymi w [5] ustaliliśmy 0.8 dla prawdziwych i 0.1 dla fałszywych. Zgodnie z [4] zarówno generator jak i dyskryminator wykorzystują optimizer Adama zaimplementowany w pakiecie PyTorch z parametrami `lr`, `beta1` i `beta2` zdefiniowanymi jak wcześniej.

W modelu wyróżniamy zatem dwie funkcje straty:

- `real_loss` liczącą stratę dla etykiet prawdziwych, tj. $y_n = 0.8$
- `fake_loss` liczącą stratę dla etykiet fałszywych, tj. $y_n = 0.1$

Trening

Trening modelu jest przeprowadzany w dwóch częściach. W pierwszej aktualizujemy dyskryminator, w drugiej generator.

Parametry

- `noise_size` - rozmiar początkowego wektora szumu
- `img_size` - rozmiar obrazów wykorzystywanych w treningu
- `lr` - szybkość uczenia się dla treningu, zgodnie z [4] powinna wynosić 0.0002
- `beta1` - parametr β_1 dla optimizera Adama, zgodnie z [4] powinna wynosić 0.5
- `beta2` - parametr β_2 dla optimizera Adama, przyjmujemy domyślną wartość 0.999
- `EPOCHS` - liczba epok treningu
- `sample_size` - rozmiar próbki obrazów dla każdej epoki

Trening dyskryminatora

Dla przypomnienia, celem treningu dyskryminatora jest zmaksymalizowanie prawdopodobieństwa poprawnego zaklasyfikowania danego obrazu jako prawdziwego (tj. pochodzącego ze zbioru treningowego) lub fałszywego. W związku z tym chcemy maksymalizować $\log(D(x)) + \log(1 - D(G(z)))$. Będziemy to robić w trzech krokach.

Na początek skonstruujemy zbiór próbek prawdziwych danych, na których wywołamy D , obliczymy stratę dla etykiet prawdziwych (`d_real_loss`). Następnie przy pomocy generatora wygenerujemy z szumu o danej wielkości `size` próbkę fałszywych obrazów i policzymy stratę dyskryminatora dla fałszywych etykiet (`d_fake_loss`). W ostatnim kroku dodamy otrzymane wartości `d_real_loss` i `d_fake_loss` do siebie otrzymując łączną stratę `d_loss` i wyliczymy jej gradienty.

Powyższy algorytm podsumowano przy pomocy pseudokodu:

Algorithm 2 Trening dyskryminatora

```
1: zainicjuj train_generator(optimizer, real_images, size)
2: wyzeruj gradienty optimizera
3: d_real  $\leftarrow D(\text{real\_images})$ 
4: d_real_loss  $\leftarrow \text{real\_loss}(\text{d\_real})$ 
5: z  $\leftarrow$  szum rozmiaru size
6: fake_images  $\leftarrow G(z)$ 
7: d_fake  $\leftarrow G(z)$ 
8: d_fake_loss  $\leftarrow \text{fake\_loss}(\text{d\_fake})$ 
9: d_loss  $\leftarrow \text{d\_real\_loss} + \text{d\_fake\_loss}$ 
10: oblicz gradient d_loss
11: zaktualizuj parametry d_optim
12: return d_loss
```

Trening generatora

Generator trenujemy poprzez minimalizowanie $\log(1 - D(G(z)))$ w celu generowania lepszych obrazów. W praktyce, jak opisano w [3] lepiej jest maksymalizować $\log(D(G(z)))$. W pierwszej kolejności generujemy wektor szumu danego rozmiaru `size` i na jego podstawie korzystając z generatora tworzymy próbkę fałszywych obrazów. Następnie klasyfikujemy wygenerowane obrazy przy pomocy dyskryminatora i na podstawie wyniku wyliczamy funkcję straty dla etykiet prawdziwych (`g_loss`). Na koniec wyliczamy gradienty.

Algorithm 3 Trening generatora

```
1: zainicjuj train_generator(optimizer, size)
2: wyzeruj gradienty optimizera
3: z  $\leftarrow$  szum rozmiaru size
4: fake_images  $\leftarrow G(z)$ 
5: d_fake  $\leftarrow D(\text{fake\_images})$ 
6: g_loss  $\leftarrow \text{real\_loss}(\text{d\_fake})$ 
7: oblicz gradient g_loss
8: zaktualizuj parametry g_optim
9: return g_loss
```

5 Wyniki

W modelu DCGAN wygenerowane obrazy są tworzone z szumu, to znaczy wektora o wartościach losowych pochodzących z pewnego rozkładu. Jednym z pytań jakie postawiliśmy sobie w trakcie przygotowywania projektu było, jak zastosowanie różnych rozkładów przy generowaniu pierwotnego szumu wpłyną na końcowe wyniki. W związku z tym zastosowaliśmy kilka rozkładów: jednostajny, normalny oraz gamma.

Porównanie wyników dla różnych rozkładów

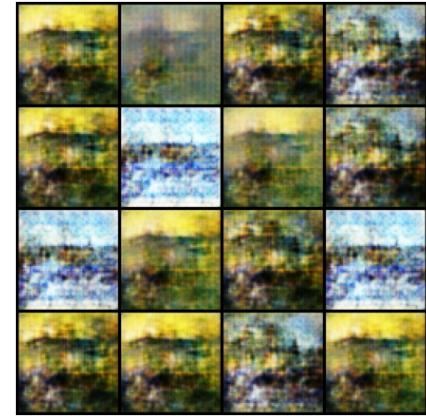
Poniżej zaprezentowano kilka wyników z wykorzystaniem różnych rozkładów do generowania szumu. Dla każdego przypadku trening został przerwany po 400 iteracjach.



Rysunek 7: Obrazy wygenerowane z rozkładu jednostajnego $U(-1, 1)$



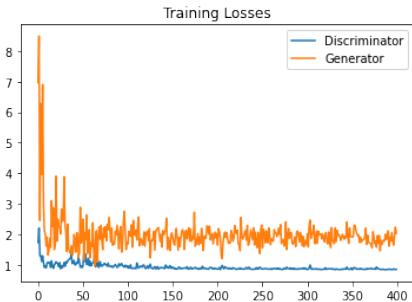
Rysunek 8: Obrazy wygenerowane z rozkładu normalnego $N(0, 1)$



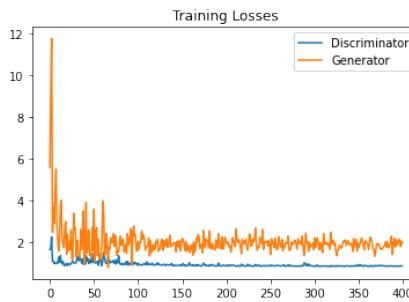
Rysunek 9: Obrazy wygenerowane z rozkładu gamma $\text{Gamma}(2, 1)$

Widzimy, że wyniki dla rozkładu gamma odstają od pozostałych. W przypadku tego rozkładu wygenerowane obrazy często się powtarzają i można stwierdzić, że dla ludzkiego oka są najmniej podobne do prawdziwych obrazów Moneta.

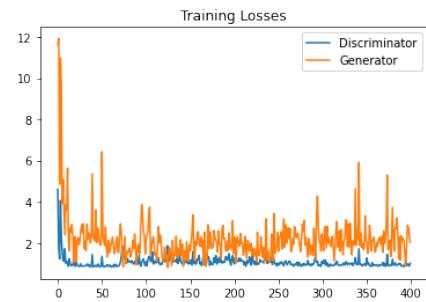
Poniżej zaprezentowano, jak zmieniały się straty w trakcie treningu dla poszczególnych rozkładów.



Rysunek 10: Straty podczas treningu dla rozkładu jednostajnego $U(-1, 1)$



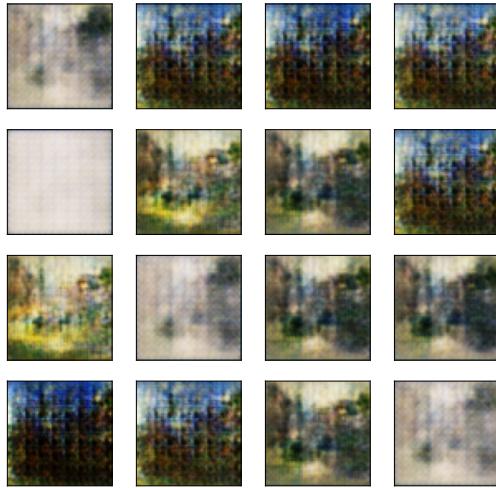
Rysunek 11: Straty podczas treningu dla rozkładu normalnego $N(0, 1)$



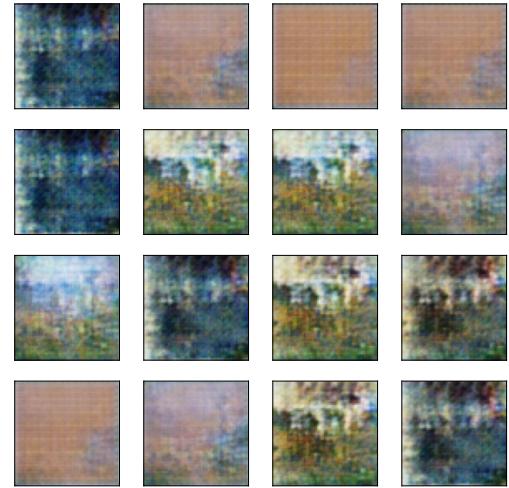
Rysunek 12: Straty podczas treningu dla rozkładu gamma $\text{Gamma}(2, 1)$

Eksperymenty z rozkładem normalnym

Na przykładzie rozkładu normalnego postanowiliśmy sprawdzić, jak zmiana parametrów rozkładu wpłynie na wyniki końcowe. Na początek postanowiliśmy zmienić wartość średniej.



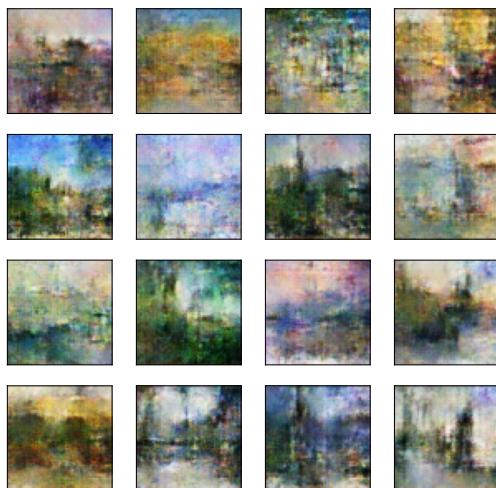
Rysunek 13: Obrazy wygenerowane z rozkładu normalnego $N(-1, 1)$



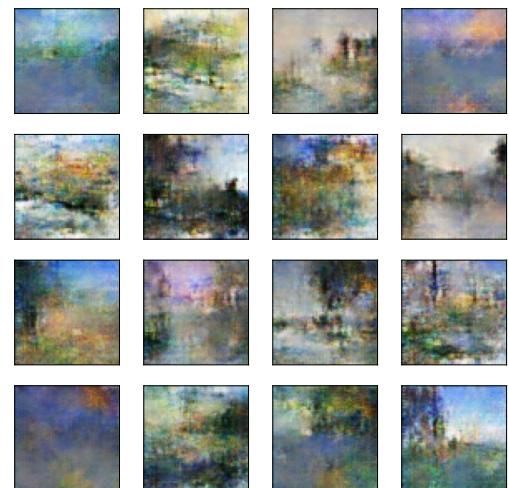
Rysunek 14: Obrazy wygenerowane z rozkładu normalnego $N(1, 1)$

Widzimy, że podobnie jak w przypadku rozkładu gamma wygenerowane obrazy charakteryzuje duża powtarzalność w próbce.

Następnie zmieniliśmy wartość wariancji.



Rysunek 15: Obrazy wygenerowane z rozkładu normalnego $N(0, 0.01)$



Rysunek 16: Obrazy wygenerowane z rozkładu normalnego $N(0, 100)$

W obu przypadkach wyniki wydają się dość zbliżone do wyników dla standardowego rozkładu normalnego, chociaż w przypadku większej wariancji wygenerowane obrazy są bardziej rozmazane.

6 Podsumowanie i wnioski

W ramach projektu udało nam się zaimplementować sieć DCGAN oraz przeprowadzić eksperymenty, badające wpływ wyboru rozkładu prawdopodobieństwa do generowania szumu na jakość generowanych obrazów. Na podstawie otrzymanych wyników wysunęliśmy następujące wnioski:

- Najlepsze wyniki otrzymaliśmy wykorzystując symetryczne rozkłady, zwracające w większości wartości z przedziału (-1, 1). Takim rozkładami są na przykład rozkład jednostajny $U(-1, 1)$ lub standardowy rozkład normalny $N(0, 1)$.
- Dla rozkładu normalnego zmiana wartości oczekiwanej może powodować znaczne pogorszenie wyników. Zmiana wariancji nie ma aż tak dużego wpływu, ale różnice w porównaniu ze standardowym rozkładem normalnym są wciążauważalne.
- Wytrenowanie dobrego modelu do generowania obrazów wymaga dużej mocy obliczeniowej. Zwiększając liczbę epok treningu być może moglibyśmy poprawić otrzymane wyniki, musimy się jednak mierzyć z ograniczeniami sprzętowymi.

7 Źródła

- [1] "I'm Something of a Painter Myself", Kaggle, <https://www.kaggle.com/c/gan-getting-started>
- [2] "DCGAN Tutorial", *PyTorch*,
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
- [3] Goodfellow, Ian & Pouget-Abadie, Jean & Mirza, Mehdi & Xu, Bing & Warde-Farley, David & Ozair, Sherjil & Courville, Aaron & Bengio, Y.. (2014). Generative Adversarial Networks. Advances in Neural Information Processing Systems. 3. 10.1145/3422622.
- [4] Radford, Alec & Metz, Luke & Chintala, Soumith. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.
- [5] Brownlee, Jason. "Tips for Training Stable Generative Adversarial Networks", *Machine Learning Mastery*, 19 czerwca 2019, <https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/>
- [6] Byeon, Eunjoo. "Exploratory Data Analysis Ideas for Image Classification", *Towards Data Science*, 11 wrzesień 2020, <https://towardsdatascience.com/exploratory-data-analysis-ideas-for-image-classification-d3fc6bbfb2d2>
- [7] Cieślik, Jakub. "How to Do Data Exploration for Image Segmentation and Object Detection (Things I Had to Learn the Hard Way)", *neptune.ai*, 16 października 2016, <https://neptune.ai/blog/data-exploration-for-image-segmentation-and-object-detection>
- [8] Agarwal, Rahul. "An End to End Introduction to GANs", *Towards Data Science*, 15 czerwca 2019, <https://towardsdatascience.com/an-end-to-end-introduction-to-gans-bf253f1fa52f>