Raport projektu #2

z przedmiotu Metody Głębokiego Uczenia

Piotr Podbielski, grupa nr 3

18 kwietnia 2019

Streszczenie

Raport opisuje zbiór danych oraz metody oraz architektury użyte do rozwiązania problemu generowania obrazów na podstawie zbioru *CIFAR-10*.

1 Wstęp

Niniejszy dokument powstał w wyniku drugiego projektu z przedmiotu *Metody Głębokiego Uczenia* wykładanego na kierunku *Inżynierii i Analizy Danych* w roku akademickim *2018/2019* na wydziale *Matematyki i Nauk Informacyjnych Politechniki Warszawskiej*.

Celem głównym projektu jest zaimplementowanie modelu *GAN* na podstawie artykułu Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville und Bengio (2014), który umożliwi generowanie obrazów na podstawie zbioru *CIFAR-10*. Implementacja musi spełniać następujące warunki:

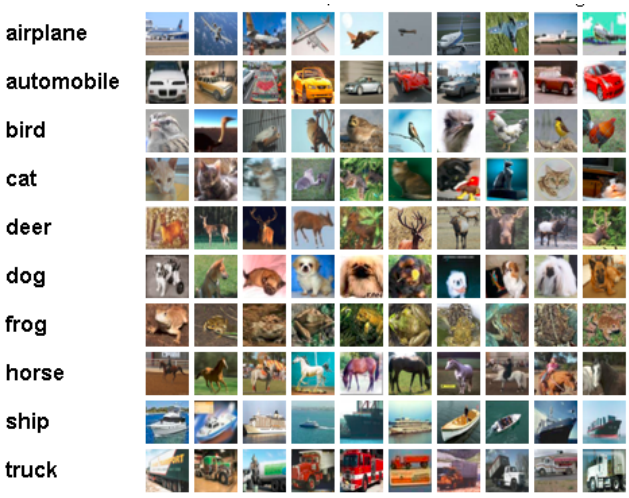
- udostępnianie możliwości wyboru architektury modułów generatora i dyskryminatora w wariantach:
  - generator i dyskryminator są sieciami w pełni połączonymi,
  - generator i dyskryminator są sieciami, odpowiednio, konwolucyjną oraz "dekonwolucyjną",
- wizualizowanie funkcji kosztu podczas trenowania, oddzielnie dla modułu generatora i dyskryminatora,
- wizualizowanie obrazów generowanych przez moduł generatora w zależności od epoki,
- wizualizowanie par: wygenerowany obraz, obraz jemu najbliższy ze zbioru treningowego w przestrzeni pikseli,
- wizualizowanie obrazów uzyskanych przez interpolację przestrzeni liniowej.

Celami dodatkowymi, w przypadku wykonania projektu przed czasem, są:

- zastosowanie architektury *DC-GAN*,
- implementacja metody *Wasserstein'a*.

2 Zbiór danych CIFAR-10

Zbiór danych uczących i walidacyjnych dostępny jest na stronie Alex'a Krizhevsky'ego. Zbiór treningowy składa się z 50 tyś. przykładów, zaś walidacyjny z 10 tyś., aczkolwiek na potrzeby architektury *GAN* zbiory te zostaną połączone a modele zostaną wytrenowane na prawie całej części połączonych zbiorów. Każdy z przykładów jest 3-kanałowym obrazem o rozmiarach 32x32 pikseli wraz z klasą, do której należy. W przypadku bezwarunkowego modelu *GAN* informacje o klasie nie są potrzebne, dlatego podczas konstrukcji modelu zostaną porzucone. Obrazy te przedstawiają środki lokomocji oraz zwierzęta. Przykładowy podzbiór przykładów widoczny jest na rysunku 1.



Rysunek 1: Przykładowy podzbiór przykładów ze zbioru *CIFAR-10*.

Każdy piksel obrazu reprezentują trzy wartości będące odwzorowaniem nasycenia w każdym z podstawowych kolorów (R, G, B). Wartości nasycenia pikseli zawierają się w skali od 0 do 255.

3 Opis użytych metod

3.1 Przeskalowanie danych wejściowych

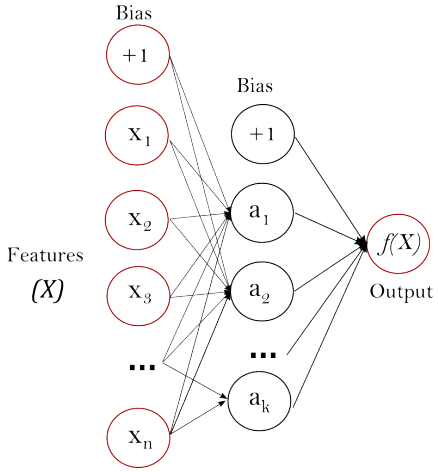
Sieci neuronowe osiągają zbieżność szybciej, gdy dane wejściowe są znormalizowane. Standaryzacja danych oznacza, iż ich wartość średnia jest równa 0, a wariancja jest równa 1, dzięki czemu większość wartości znajduje się w przedziale od -1 do 1.

Zamiast dokonywania standaryzacji, w implementacji dokonano przeskalowania danych wejściowych. Początkowo wartości nasycenia pikseli były z przedziału od 0 do 255, a po przeskalowaniu są one z przedziału od -1 do 1. Uzyskano to dzięki odjęciu od wartości nasycenia pikseli liczby 128 a następnie podzielenie przez liczbę 128.

3.2 Podstawowe rodzaje sieci

3.2.1 Perceptron wielowarstwowy (ang. *Multilayer perceptron*)

Perceptron wielowarstwowy jest typem sieci neuronowej z przepływem informacji w przód (ang. *Feedforward Neural Network*), co oznacza, że nie występują w niej pętle zwrotne. Podczas przepływu informacji *w przód*, w warstwie i wykorzystywane są tylko wartości aktywacji z warstwy $i-1$. W warstwach ukrytych *MLP* wartość aktywacji każdego neurona obliczana jest na podstawie kombinacji liniowej wszystkich wartości aktywacji neuronów z warstwy poprzedniej. Przykładowa architektura perceptronu wielowarstwowego została przedstawiona na rysunku 2.



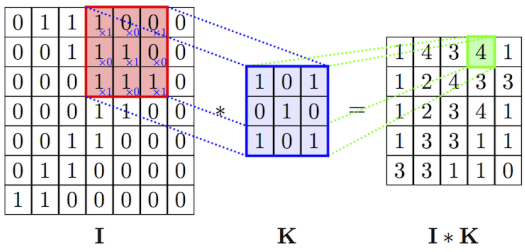
Rysunek 2: Przykładowa architektura perceptronu wielowarstwowego z jedną warstwą ukrytą. Źródło.

3.2.2 Sieć konwolucyjna (ang. *Convolutional Neural Network*)

Sieci konwolucyjne są także typem sieci neuronowej z przepływem informacji w przód (ang. *Feedforward Neural Network*). Różnica w stosunku do *MLP* jest taka, że w sieciach konwolucyjnych dzięki użyciu splotu funkcji wartość aktywacji dla każdego z neuronów w warstwach ukrytych wyliczana jest na podstawie części neuronów z warstw poprzednich. Dzieje się to w taki sposób, że nie każdy neuron w warstwie ukrytej ma połączenie z wszystkimi neuronami z warstwy poprzedniej.

W przypadku obrazów zastosowanie konwolucji polega na przesuwaniu okna o pewnym rozmiarze po neuronach w warstwie i i zastosowywaniu na tych neuronach splotu z pewnym filtrem, który reprezentują wagi warstwy ukrytej. Filtr ten podlega nauce przez sieć podczas procesu trenowania za pomocą propagacji wstecznej.

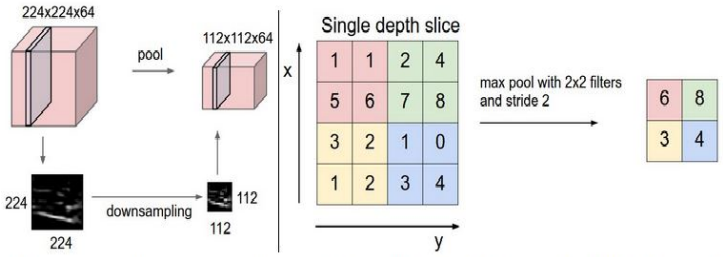
Przykład działania konwolucji (splotu) w sieciach konwolucyjnych został przedstawiony na rysunku 3.



Rysunek 3: Przykład działania funkcji konwolucji dwuwymiarowej, która oblicza wartość neuronu w kolejnej warstwie na podstawie części neuronów z poprzedniej warstwy oraz zadanego filtru. Źródło.

Każda z warstw konwolucji w *CNN* składa się z filtru, który przesuwany jest po neuronach w warstwie poprzedniej co pewną wartość *stride*. Dodatkowo neurony w warstwie poprzedniej mogą zostać wzbogacone o sztuczne neurony zwane *padding'iem*. W zależności od *stride'u*, *padding'u* i kilku innych parametrów konwolucji uzyskujemy różną liczbę neuronów wyjściowych w warstwie konwolucji. *Padding* stosujemy wtedy, gdy chcemy, aby liczba neuronów w warstwie poprzedniej była równa liczbie neuronów na wyjściu konwolucji.

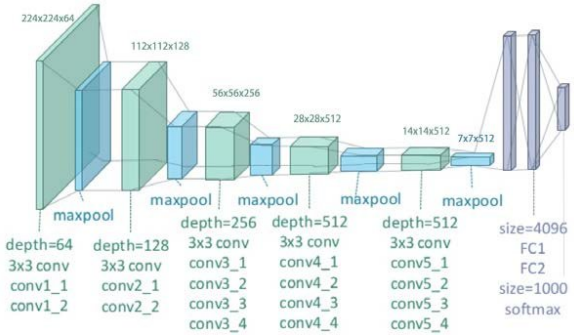
Do każdej z warstw konwolucji dołączyć można warstwę zwaną *pooling'iem*. *Pooling* dwuwymiarowy ma za zadanie drastycznie zmniejszyć liczbę neuronów w każdej kolejnej warstwie sieci konwolucyjnej poprzez przejście pewnym zadanym oknem po neuronach warstwy poprzedniej i zagregowanie wartości z tego okna do jednej, która razem z wartościami z reszty okien tworzy wyjście z warstwy *pooling'u*. Przedstawiono to na obrazku 4.



Rysunek 4: Przykład działania *pooling'u* w sieciach konwolucyjnych. Źródło.

3.2.3 Głęboka sieć konwolucyjna (ang. *Deep Convolutional Neural Network*)

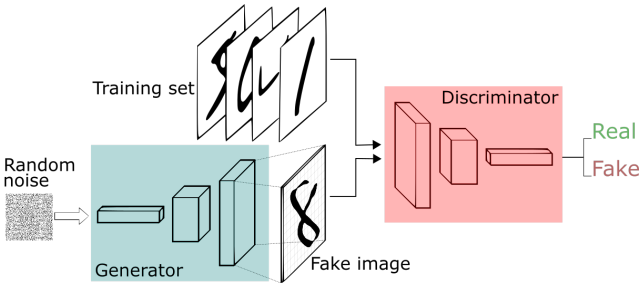
Głębokie sieci konwolucyjne są odmianą sieci konwolucyjnych opisanych w rozdziale 3.2.2, z dodatkowym założeniem, że muszą mieć wiele warstw ukrytych (często dziesiątki). Sieci takie potrafią mieć po kilkadziesiąt albo kilkaset milionów parametrów (parametry są to po prostu wagi, które są uczone przez sieć w trakcie trenowania). Przykładem głębokiej CNN jest architektura *VGG-19* przedstawiona na rysunku 5. Architektura ta ma 144 milionów parametrów.



Rysunek 5: Przykład głębokiej sieci konwolucyjnej *VGG-19*. Źródło.

3.2.4 Sieć generatywna (ang. *General Adversarial Networks*

GAN są dosyć nowymi sieciami, bo wymyślonymi w 2014 r. przez *Ian'a Goodfellow'a*. *GAN'y* są modelami generatywnymi i jest to swego rodzaju gra dwóch sieci (zwanych dalej modułami), które konkurują ze sobą. Moduł generatywny (ang. *generative*) odpowiedzialny jest za generowanie pewnej wyuczonej przez moduł reprezentacji z zadanego wektora wejściowego (przeważnie losowego, będącego szumem). Drugi zaś moduł, zwany dyskryminatorem (ang. *discriminator*) odpowiedzialny jest za klasyfikację tejże wygenerowanej reprezentacji, oraz reprezentacji ze zbioru treningowego w dwóch klasach: czy (1) dana reprezentacja jest prawdziwa, czy może (2) fałszywa. Trenowanie *GAN'ów* opiera się na przemiennym optymalizowaniu funkcji kosztu obu modułów. Schemat koncepcji *GAN* został przedstawiony na rysunku 6.



Rysunek 6: Koncepcja *GAN*, zaproponowana przez *Ian'a Goodfellow'a*, na przykładzie danych dwuwymiarowych. Źródło.

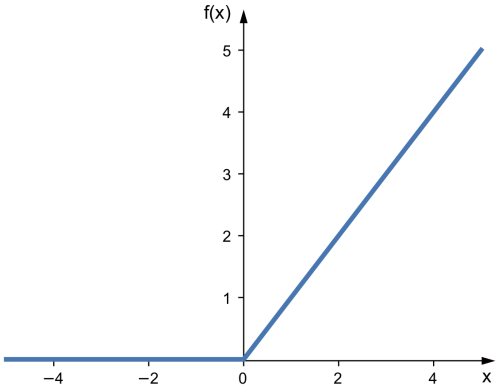
Kwestia architektury obu modułów jest kwestią wtórną. Zazwyczaj są to głębokie sieci konwolucyjne (np. *DC-GAN*).

3.3 Funkcje aktywacji

Aby sieć neuronowa mogła aproksymować funkcje, w których zmienna objaśniana jest nieliniową zależnością zmiennych objaśniających, należy wprowadzić do sieci pewien element nieliniowości. Elementem nieliniowości w sieciach neuronowych są funkcje aktywacji. Funkcje te, aby spełniać swoją rolę, muszą być nieliniowe.

3.3.1 ReLU

Jedną z nieliniowych funkcji aktywacji użytych w projekcie jest funkcja *ReLU* (ang. *Rectified Linear Unit*). Wyraża się ona wzorem $f(x) = max(0, x)$. Jej charakterystykę przedstawiono na rysunku 7.



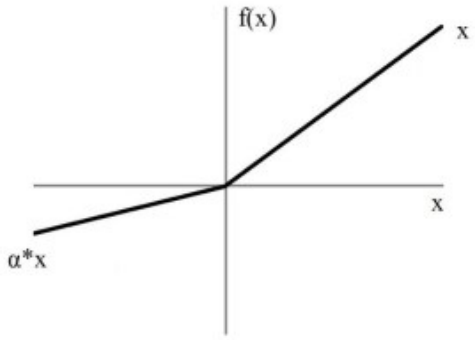
Rysunek 7: Funkcja aktywacji ReLU. Źródło.

Jest to jedna z najprostszych funkcji nieliniowych, a zarazem najczęściej używanych i sprawdzających się w sieciach neuronowych.

3.3.2 LeakyReLU

Modyfikacją funkcji aktywacji *ReLU* jest funkcja *Leaky ReLU*. W przypadku użycia funkcji aktywacji *ReLU*, gdy wartość $x < 0$, to podczas propagacji wstecznej gradient dla tej wartości jest równy 0. *Leaky ReLU* pozwala na skorygowanie tej właściwości i propaguje pewien gradient nawet wtedy, gdy wartość $x < 0$. Wzór funkcji aktywacji *LeakyReLU* przedstawiono poniżej (1), a zobrazowano ją na rysunku 8.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{w p. p.} \end{cases} \tag{1}$$



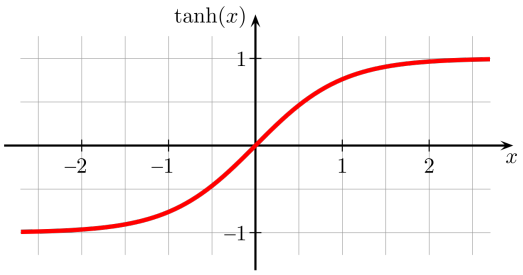
Rysunek 8: Funkcja aktywacji Leaky ReLU. Źródło.

3.3.3 tanh

Funkcja aktywacji tangensa hiperbolicznego (*tanh*) jest funkcją, którą z reguły używa się na wyjściu z danej sieci lub modułu. Jej głównym zadaniem jest zapewnienie, aby wartości przepuszczone przez tę

funkcję aktywacji były z zakresu od -1 do 1. Jej wzór został przedstawiony poniżej (2), a zobrazowano ją na rysunku 9.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2}$$

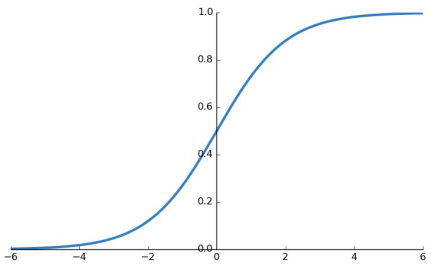


Rysunek 9: Funkcja aktywacji *tanh*. Źródło.

3.3.4 sigmoid

Funkcja *sigmoid* (zwana także logistyczną) jest również funkcją, którą zazwyczaj używa się na końcu sieci neuronowej. Z reguły używa się jej do modelowania prawdopodobieństwa występowania danej zmiennej losowej, gdyż jej wartości są z przedziału od 0 do 1. Jej wzór został przedstawiony poniżej (3), a zobrazowano ją na rysunku 10.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$



Rysunek 10: Funkcja aktywacji *sigmoid*. Źródło.

3.4 Funkcje straty

Aby optymalizować wagi perceptronów używany jest mechanizm propagacji wstecznej gradientów. Funkcje straty użyte w architekturach tego projektu, które są minimalizowane w celu optymalizacji wag dyskryminatora oraz generatora zostały przedstawione w poniższych podpodrozdziałach.

3.4.1 Minimax loss

GAN'y są definiowane jako gra minimax z funkcją straty przedstawioną poniżej (4).

$$\min_G \max_D V(D, G) = E_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{4}$$

Optymalizowanie takiej gry sprowadza się do naprzemiennego optymalizowania wag dyskryminatora (wzór 5) oraz generatora (wzór 6).

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right] \tag{5}$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \tag{6}$$

Wartym zaznaczenia jest fakt, że podczas optymalizacji wag dyskryminatora dodajemy gradienty do macierzy wag, a podczas optymalizacji wag generatora dokonujemy klasycznego odejmowania gradientów od macierzy wag.

3.4.2 Modified minimax loss

Aby pozbyć się problemu zanikania gradientów (ang. *vanishing gradients*), twórcy artykułu o *GAN'ach* zaproponowali modyfikację sposobu aktualizacji wag generatora. Nowy wzór optymalizowania wag modułu generatora został przedstawiony poniżej (7).

$$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(z^{(i)}\right)\right)\right) \tag{7}$$

3.5 Metody regularyzacji

Aby sieci neuronowe nie ulegały przetrenowaniu, dodaje się do nich mechanizmy regularyzacji. Przetrenowanie polega na nadmiernym dopasowaniu się wag sieci neuronowej do zbioru treningowego, co skutkuje na przykład dużymi (co do modułu) wartościami wag sieci neuronowej.

3.5.1 Batch Normalization

Normalizacja batchy jest jednym z mechanizmów regularyzacji w sieciach neuronowych, który kontroluje średnią i odchylenie standardowe wartości wyjścia z danej warstwy. Sieci neuronowe osiągają szybciej zbieżność, gdy wartości wejścia do kolejnych warstw są o podobnym zakresie w każdym z wymiarów. Pomysł normalizacji danych wejściowych został przeniesiony do wnętrza sieci (warstw ukrytych) i tak powstała regularyzacja *Batch Normalization*. Sprowadza się ona do tego, że podczas treningu sieci, dany *mini-batch* jest normalizowany, ale w celu zachowania większej elastyczności następnie mnożony przez pewien parametr γ oraz sumowany z innym parametrem β. Te dwa parametry są jednakże uczone na całym zbiorze, a nie tylko w obrębie jednego *mini-batch'a*.

Wzory ukazujące sposób działania *Batch Normalization* zostały przedstawione poniżej (8).

$$\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \tfrac{1}{m} \textstyle\sum_{i=1}^{m} x_i \\
\sigma_{\mathcal{B}}^2 &\leftarrow \tfrac{1}{m} \textstyle\sum_{i=1}^{m} \left(x_i - \mu_{\mathcal{B}}\right)^2 \\
\widehat{x}_i &\leftarrow \tfrac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
y_i &\leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right)
\end{aligned} \tag{8}$$

3.5.2 Dropout

Dropout jest kolejnym z mechanizmów regularyzacji. Polega on na wyłączaniu podczas trenowania losowych neuronów z zadanym prawdopodobieństwem p. Powoduje to, że podczas trenowania sieć jest w pewien sposób ograniczona, ma mniejszą pojemność, niż wynikało by z jej bazowej architektury. Sieć o mniejszej pojemności (ang. *capacity*) pozwala na przybliżanie funkcji mniej skomplikowanych. Dzięki temu uzyskujemy efekt ograniczenia przetrenowania sieci neuronowej. Innym efektem wyłączania losowych neuronów jest to, że na predykcję rzadziej wpływ ma wartość konkretnego neuronu. Dzięki mechanizmowi dropout różne neurony muszą nauczyć się wykrywać różne końcowe cechy. Na przykładzie zbioru CIFAR oznacza to, że model będzie miał mniejszą skłonność do podejmowania decyzji o prawdziwości albo fałszywości danego obrazka z modułu generatora na przykład na podstawie wartości nasycenia jednego piksela, gdyż podczas trenowania losowe piksele zostały deaktywowane, a moduł nadal miał za zadanie nauczyć się poprawnie klasyfikować obrazki.

3.6 Optymalizator ADAptive Momentum

W celu optymalizacji wag sieci neuronowych używa się optymalizatorów. Poniżej przedstawiono jeden z nich, który został użyty w projekcie.

Optymalizator ADAM (ang. *Adaptive Momentum*) dzięki momentom 1. oraz 2. rzędu aktualizując macierze wag sieci neuronowej korzysta z historii wartości tych macierzy. Efektem tego jest możliwość

wyciszania albo wzmacniania poszczególnych gradientów. Wzór na wyliczenie macierzy gradientów przedstawiony został poniżej (9).

$$W_d = \frac{v_t}{\sqrt{s_t + \epsilon}} * g_t \tag{9}$$

Gdzie v_t to średnia wykładnicza gradientów wyrażona wzorem 10, s_t to średnia wykładnicza kwadratów gradientów wyrażona wzorem 11, a g_t to gradienty wag.

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \tag{10}$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \tag{11}$$

Gdzie β_1 i β_2 to hiperparametry średniej wykładniczej.

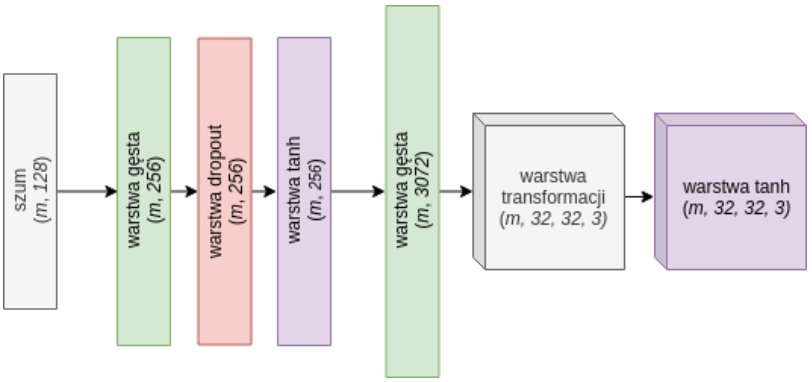
4 Architektury i hiperparametry wytrenowanych modeli

Założenia wspólne dla każdej z architektur były następujące:

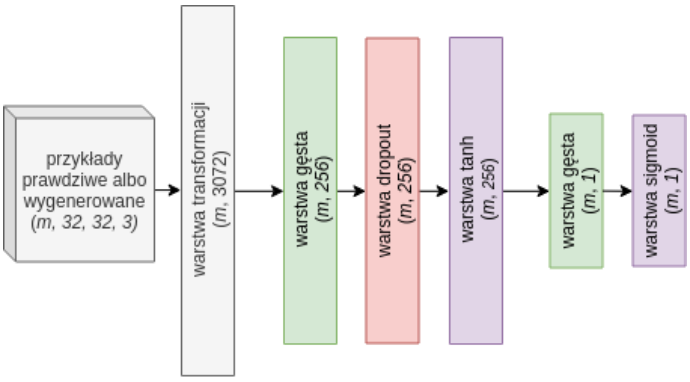
- Liczba przykładów podczas jednego przebiegu ustawiona na 1024,
- użycie 8 rdzeni *TPU*,
- uczenie modeli na 59392 przykładach ze względu na to, że liczba ta dobrze dzieli się przez rozmiar batch'a,
- użycie optymalizatora *ADAM* z hiperparametrami: α równym 0.0002, β_1 równym 0.5, β_2 pozostawionym jako domyślny,
- użycie *modified minimax loss* jako funkcji straty,
- rozmiar wektora szumu ustawiony na 128,
- trenowanie ustawione na 1000 epok.

4.1 MLP-GAN

MLP-GAN jest architekturą, której nazwa wynika z tego, że zbudowana jest z wielowarstwowych perceptronów. Występują w niej warstwy gęste, regularyzacja *dropout* oraz funkcje aktywacji *tanh* oraz *sigmoid*. Architektura modułu generatora została przedstawiona na rysunku 11, zaś modułu dyskryminatora na rysunku 12.



Rysunek 11: Architektura modułu generatora sieci *MLP-GAN*.



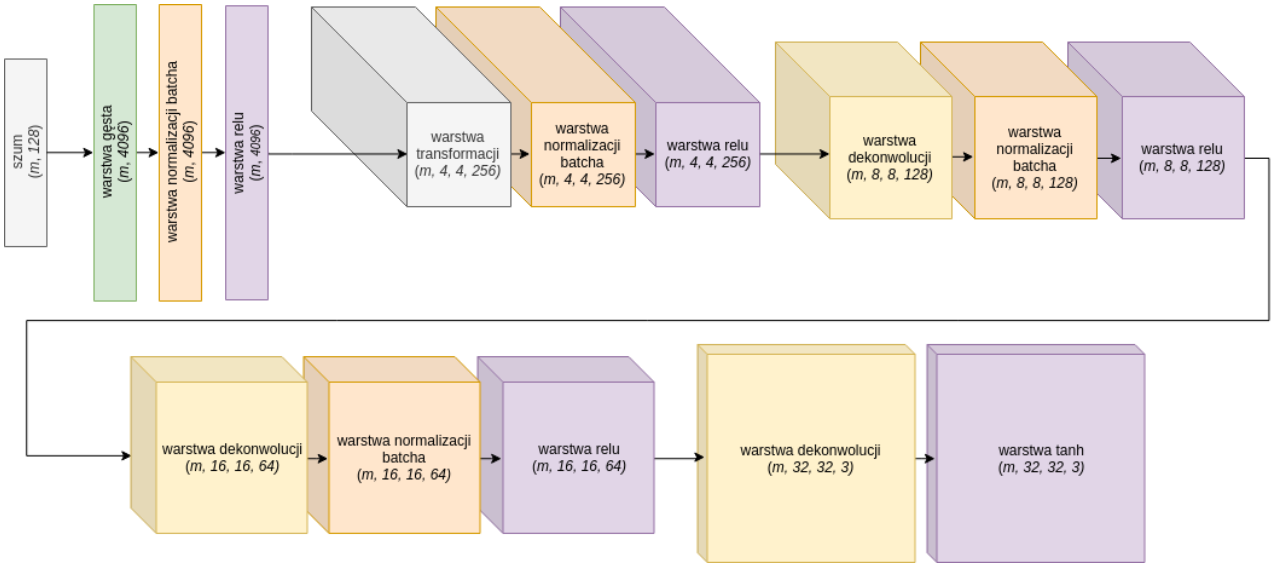
Rysunek 12: Architektura modułu dyskryminatora sieci *MLP-GAN*.

4.2 DC-GAN

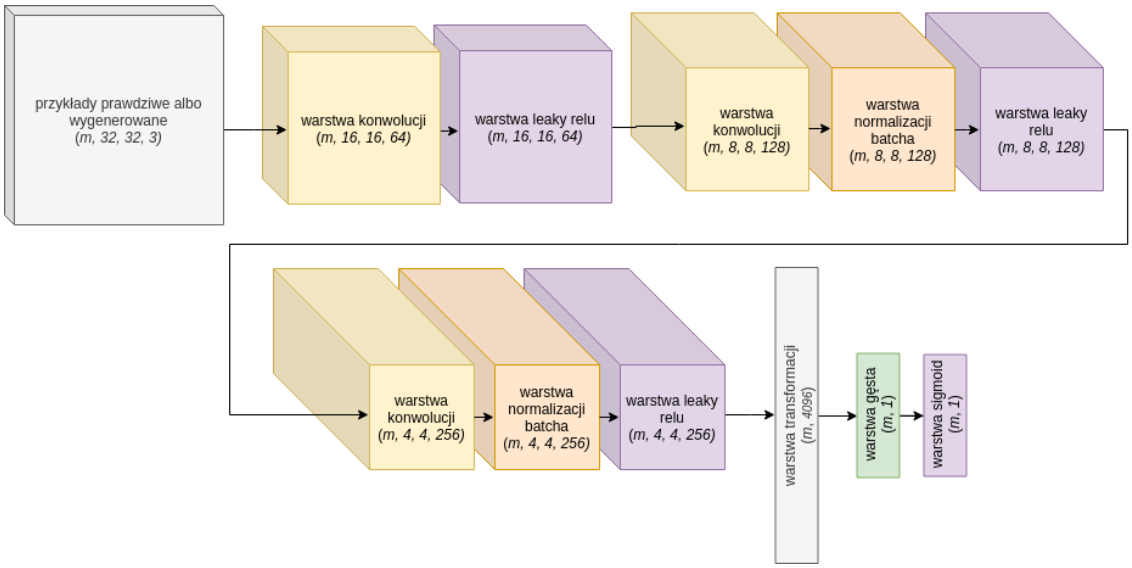
DC-GAN jest architekturą zaproponowaną w artykule Radford, Metz und Chintala (2015). Charakteryzuje się ona:

- Użyciem konwolucji ze stride'm,
- użyciem dekonwolucji (transponowanej konwolucji),
- wyeliminowaniem warstw w pełni połączonych,
- zastosowaniem normalizacji batch'a jako metody regularyzacji,
- użyciem funkcji aktywacji *ReLU* w generatorze za wyjątkiem ostatniej warstwy generatora,
- użyciem *Leaky ReLU* w dyskryminatorze.

Architektura modułu generatora została przedstawiona na rysunku 13, zaś modułu dyskryminatora na rysunku 14.



Rysunek 13: Architektura modułu generatora sieci *DC-GAN*.



Rysunek 14: Architektura modułu dyskryminatora sieci *DC-GAN*.

5 Wyniki wytrenowanych modeli

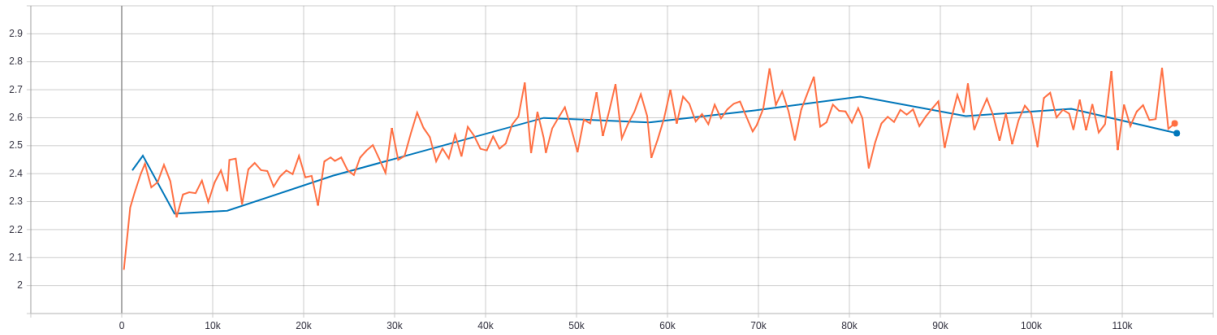
Sieci zostały wytrenowane dzięki wykorzystaniu platformy Google Colab. Na rysunku 15 przedstawiono przykładowe obrazy generowane przez sieć po 1000 epokach.

(a) (b)

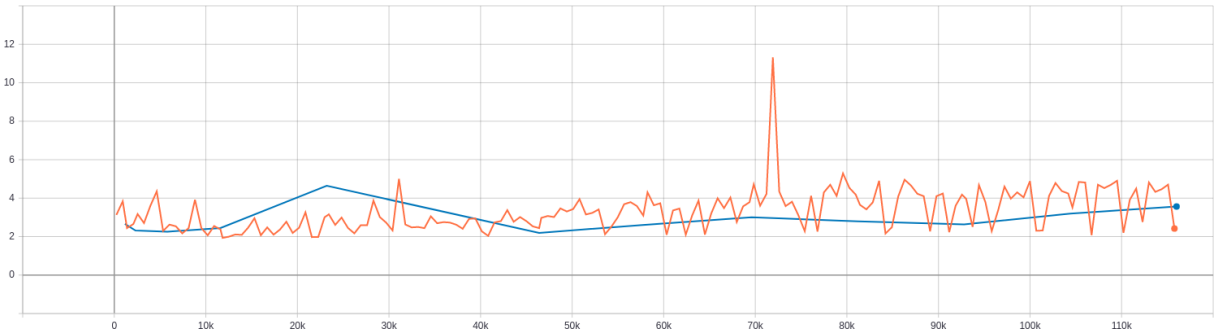
Rysunek 15: Obrazy generowane przez sieć *MLP-GAN* (a) oraz *DC-GAN* (b) po 1000 epokach trenowania.

Jak można się spodziewać, przetestowana wcześniej przez badaczy architektura *DC-GAN* sprawuje się dużo lepiej podczas nauki obrazów ze zbioru *CIFAR-10* niż wymyślona przez autora niniejszego raportu architektura *MLP-GAN*. Na przykładowych obrazach wygenerowanych przez sieć *DC-GAN* można dostrzec auta, zwierzęta i statek.

Na rysunku 16, na pomarańczowo przedstawiono wykres sumy funkcji strat w dziedzinie kroków trenowania, zaś na niebiesko sumę funkcji strat dla całego zbioru treningowego. Dwoma krokom trenowania odpowiada wytrenowanie sieci na jednym batch'u.



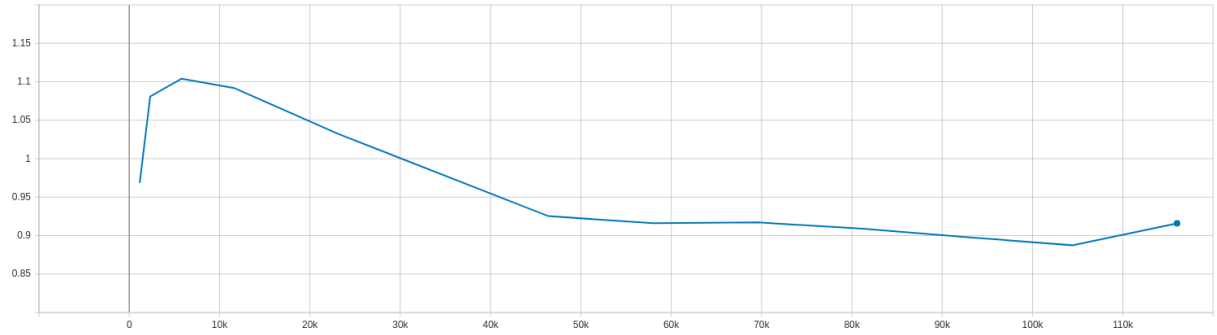
(a)



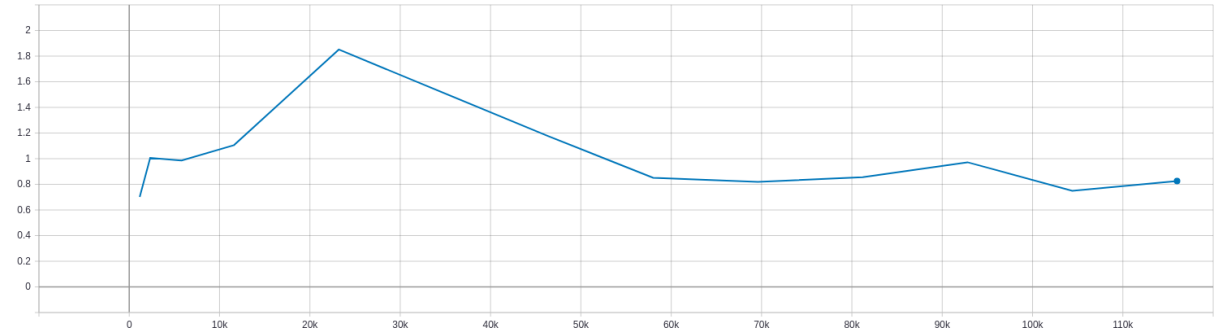
(b)

Rysunek 16: Wykresy sumy funkcji strat wyznaczonej dla sieci *MLP-GAN* (a) oraz *DC-GAN* (b).

Na rysunku 17 przedstawiono wykres funkcji straty modułu dyskryminatora w dziedzinie kroków trenowania. Ze względu na to, że podczas trenowania z wykorzystaniem API Estymatorów we frameworku `Tensorflow` na *TPU* nie jest możliwym odkładanie metryk do *Tensorboard'a*, wartości funkcji straty dla dyskryminatora zostały wyznaczane po epokach *10*, *20*, *50*, *100*, *200*, *400*, *500*, *600*, *700*, *800*, *900*, *1000* trenowania, z wykorzystaniem całego zbioru treningowego.



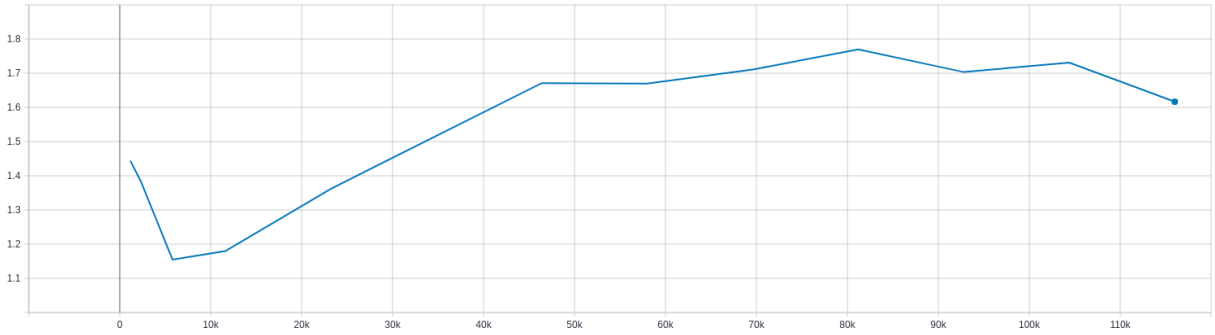
(a)



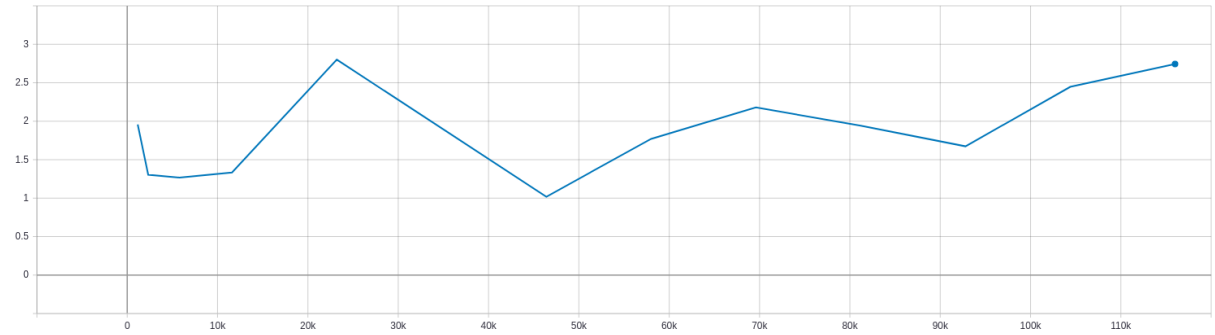
(b)

Rysunek 17: Wykresy funkcji straty modułu dyskryminatora sieci *MLP-GAN* (a) oraz *DC-GAN* (b) wyznaczane po pewnych, założonych z góry, epokach trenowania.

Na rysunku 18 przedstawiono wykres funkcji straty modułu generatora w dziedzinie kroków trenowania z takim samym zastrzeżeniem jak na rysunku powyżej.



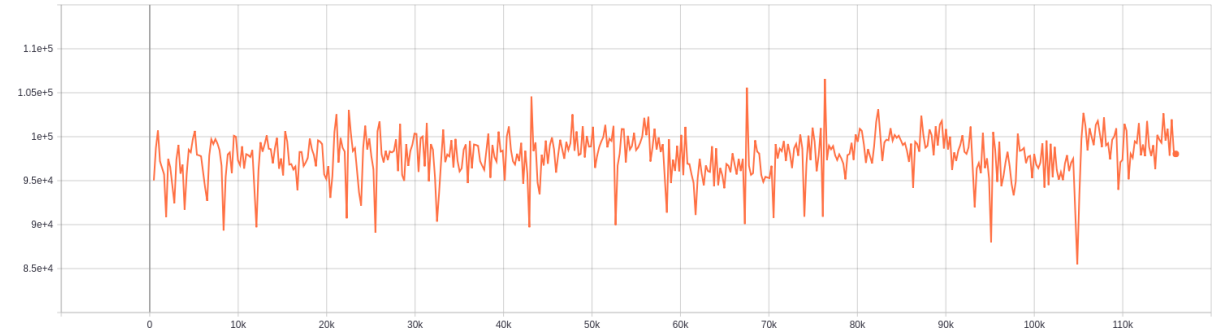
(a)



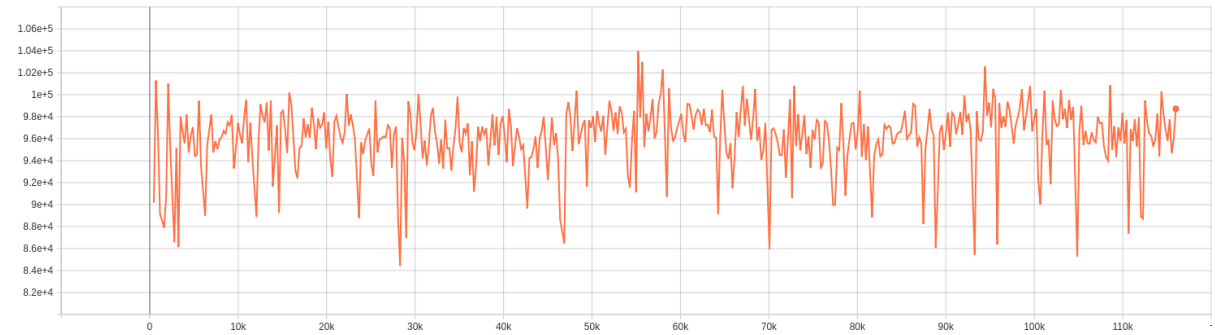
(b)

Rysunek 18: Wykresy funkcji straty modułu generatora sieci *MLP-GAN* (a) oraz *DC-GAN* (b) wyznaczane po pewnych, założonych z góry, epokach trenowania.

Na rysunku 19 przedstawiono wykres funkcji liczby przetworzonych przykładów w dziedzinie czasu.



(a)



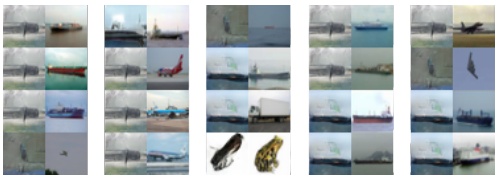
(b)

Rysunek 19: Wykres liczby przetworzonych przykładów sieci *MLP-GAN* (a) oraz *DC-GAN* (b) dla poszczególnych kroków treningu.

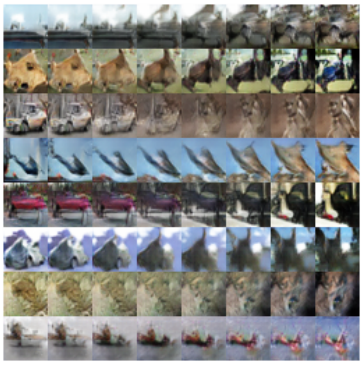
Dodatkowo, dla wytrenowanego przez 1000 epok modelu *DC-GAN*, wygenerowano:

1. 128 obrazów na podstawie losowego szumu i porównano je ze zbiorem treningowym za pomocą metody `compare_ssim` z modułu `skimage.measure` pakietu `scikit-image`. 20 par *wygenerowany obraz – przykład ze zbioru treningowego* z największym współczynnikiem podobieństwa zostało przedstawionych na rysunku 20,

2. obrazy ukazujące interpolację ukrytej przestrzeni na podstawie 16 punktów podzielonych w pary. Dla każdej z pary wyliczono 6 punktów z założeniem, że owe 6 punktów oraz 2 punkty pary ułożone są na jednej prostej w równych odstępach. Dla takich punktów wygenerowano obrazy za pomocą modułu generatora, co przedstawione jest na rysunku 21.



Rysunek 20: Obrazy 20 par *wygenerowany obraz – przykład ze zbioru treningowego* z największym współczynnikiem podobieństwa wygenerowane dla modelu *DC-GAN* trenowanego przez 1000 epok.



Rysunek 21: Obrazy wygenerowane dla 8 oktetów punktów z przestrzeni ukrytej, gdzie obrazy z każdego oktetu ułożone są na tej samej prostej w równych od siebie odległościach, wygenerowane dla modelu *DC-GAN* trenowanego przez 1000 epok.

6 Implementacja

Model został zaimplementowany w języku *Python* przy użyciu biblioteki *TensorFlow* w wersji `r1.13`. Wartości metryk przekazywane są podczas trenowania i walidacji do *TensorBoard'a*. Do porównywania obrazów użyto biblioteki *scikit-image*.

Kod dostępny jest w repozytorium na platformie GitHub.

7 Wnioski

Modele *GAN* są niezwykle trudne do trenowania. Istnieje wiele technik poprawiających stabilność treningu tych sieci, jednakże z powodu braku czasu nie przetestowano ich. W przyszłości, autor tego raportu chciałby zaimplementować metodę *Wasserstein GAN* wraz z karą za gradient (*WGAN-GP*), która poprawia stabilność treningu sieci.

Skorzystanie z implementacji sieci *GAN* w postaci Estymatora, który może być uruchomiony na *TPU* (`TPUGanEstimator`) nie było dobrą decyzją. Więcej swobody implementacyjnej dałoby zaimplementowanie estymatora samodzielnie na podstawie dostępnych funkcji z modułu `tf.contrib.gan`. W ramach wolnego czasu kod dostarczony wraz z tym raportem niewątpliwie zostanie przepisany na wersję bardziej elastyczną.

Literatura

[Goodfellow u. a. 2014] Goodfellow, Ian J. ; Pouget-Abadie, Jean ; Mirza, Mehdi ; Xu, Bing ; Warde-Farley, David ; Ozair, Sherjil ; Courville, Aaron ; Bengio, Yoshua: Generative Adversarial Networks. In: *arXiv e-prints* (2014), Jun, S. arXiv:1406.2661

[Radford u. a. 2015] Radford, Alec ; Metz, Luke ; Chintala, Soumith: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In: *arXiv e-prints* (2015), Nov, S. arXiv:1511.06434