



# Zadanie 05


## Czy sieci neuronowe śnią o ciastach marchewkowych?

W tym tygodniu nasze zadanie będzie nieco bardziej złożone. Zbudujemy nie jedną, a dwie sieci neuronowe - wzajemnie ze sobą rywalizujące i w efekcie sukcesywnie zwiększające swoje możliwości. Pierwsza z nich będzie generować możliwe realistyczne obrazy pasujące do pewnego zbioru. Druga - odróżniać tak stworzone podróbki od rzeczywistych fotografii.

- W przypadku punktów oznaczonych ikoną  poinformuj w jaki sposób je zrealizowałeś - wspomnij kluczowe klasy/metody/funkcje lub załącz powiązany fragment kodu źródłowego.
- W przypadku punktów oznaczonych ikoną  załącz w raporcie obraz przedstawiający efekt danej operacji.

## To chyba nie jest oryginalny Breitling, prawda?

Zacznijmy od prostszej (a przynajmniej bardziej podobnej do tego co robiliśmy do tej pory) z nich - dyskryminatora. Jego zadaniem jest rozpoznawać prawdziwe obrazy i eliminować te sztucznie wygenerowane - a więc mamy do czynienia z najzwyklejszą klasyfikacją! Czy w takim razie wystarczy "pożyczyć" architekturę sieci konwolucyjnej z poprzednich zadań? Nie do końca: poza skutecznym wykrywaniem fałszerstw chcemy też by na bazie jej pracy możliwe było nauczanie się lepszych sposobów generacji fałszywek. A to z kolei oznacza, że musimy zadbać również o pewny i stabilny przepływ gradientu do jej najgłębszych warstw (nawet kosztem skuteczności).

1. Przygotuj model dyskryminatora w oparciu o opisaną niżej architekturę. 
  1. Model powinien przyjmować na wejście kolorowe fotografie w rozmiarze 64x64.
  2. Następnie stosujemy następujące po sobie warstwy konwolucyjne.
    1. Powinny wykorzystywać filtr o rozmiarze 4x4, ze *stride* równym 2 i *paddingiem* typu *same* (w tym przypadku nie możemy sobie pozwolić na stratę informacji wywołaną operacją *max pooling*, więc rozmiar kanałów będziemy zmniejszali bezpośrednio wykorzystując konwolucje - stąd taki właśnie kształt kernela i *stride*).
    2. Wyjście z takiej warstwy przetwarzamy z użyciem operacji *batch normalisation*.
    3. Na koniec wykorzystujemy aktywację *leaky ReLU* z nachyleniem części negatywnej wynoszącym 0.2 (znów - agresywnie walczymy z zanikiem gradientu).
    4. Zastosujemy kolejno 3 takie warstwy, zawierające odpowiednio 64, 128 i 128 jednostek (i zwracające tyleż samo kanałów wynikowych). Sieć jest płytsza niż ta używana w poprzednich tygodniach - zależy nam zarówno na tolerowalnej wydajności obliczeń jak i na dobrym dostępie do gradientu.
  3. Na koniec spłaszczmy wszystko do jednego wektora, zastosujmy niewielki *dropout* (około 20%) i kończymy całość warstwą gęsto połączoną o jednym wyjściu aktywowanym funkcją *sigmoid* - to wyjście będzie zwracało nam prawdopodobieństwo, że dany obraz został sztucznie wygenerowany (lub nie).

## Skoro nie widać różnicy..?

Teraz przygotujemy drugą z sieci - tą tworzącą "z niczego" nowe obrazy. Cóż - tak naprawdę niezupełnie "z niczego" - wejściem do niej będzie wektor losowych wartości używany jako baza na której zbudowane zostanie potem wyjście. Zapewnia to różnorodność efektów pracy generatora oraz pozwala na pewną kontrolę nad zwracanymi rezultatami.

2. Przygotuj model generatora. Tym razem architektura prezentuje się nieco inaczej niż przywykliśmy. [📄]

1. Zaczynij od wejścia będącego wektorem składającym się ze 128 wartości.
2. To zdecydowanie za mało na najprostszy nawet obraz - musimy więc tą informację "rozmnóżać" - ale w kontrolowany sposób. Zaaplikujmy warstwę gęsto połączoną o liczbie wyjść wystarczającej, by stworzyć 128 kanałów 8x8. Będzie więc miała  $8 * 8 * 128 = 8192$  wyjść! Nie stosujemy żadnej aktywacji (ta transformacja będzie w pełni liniowa).
3. Przekształćmy uzyskany wektor tak, by stał się tensorem wspomnianych 128 kanałów 8x8 (przechodzimy ze świata wektorowego do świata przetwarzania obrazów).
4. Teraz sukcesywnie zwiększamy rozdzielczość takiego proto-obrazu, równocześnie pozwalając generatorowi na nauczanie się coraz bardziej skomplikowanych cech.
  1. W tym celu wykorzystamy warstwy konwolucji transponowanych.
  2. Niech ich rozmiar kernela również wynosi 4x4, a *stride* 2 (padding typu *same*): każde zastosowanie takiej warstwy dwukrotnie zwiększy rozdzielczość przetwarzanych kanałów. Jako aktywacji znów użyjemy *leaky ReLU* (z nachyleniem 0.2 w części negatywnej).
  3. Zastosujmy trzy takie warstwy, kolejno po 128, 256 i 512 filtrów.
5. Na tym etapie osiągnęliśmy już pożądaną rozdzielczość (64x64), ale mamy zbyt wiele kanałów (512) i niewłaściwy rozkład wynikowych wartości. Dorzućmy jedną końcową warstwę zwykłej konwolucji: o 3 kanałach wynikowych (R, G i B), rozmiarze kernela 5x5, standardowym *stride* równym 1, *paddingu* typu *same* i aktywacji *sigmoid* (by wyniki były z zakresu od 0 do 1).
6. Zauważ, że ten model to w pewnym sensie odwrotność poprzedniej sieci konwolucyjnej - przechodzimy z wektora na obraz (a nie z obrazu na wektor)!

3. Podajmy na wejście modelu losowy szum (najlepiej o rozkładzie normalnym, średniej w zerze i odchyleniu standardowym  $\pm 1$ ) i wyplotujmy wynikową fotografię. Jak wygląda? [🖼️]

## I zebrano jeszcze dwanaście koszy ułomków

Mamy już przygotowane obie sieci, teraz należy zorganizować zbiór danych, który stanie się "inspiracją" do generowania nowych obrazów. W tym celu możemy wykorzystać jedną z kategorii zgromadzonych na rzecz poprzedniego zadania (lub zebrać nowy zbiór danych korzystając z DuckDuckGo w ten sam sposób co ostatnio). Warto jednak dodatkowo zadbać o to, by nie znalazły się w nim "wadliwe" elementy - mocno utrudni to trening generatora. Dobrze, by klasa którą reprezentuje zbiór nie była zbyt różnorodna w swojej naturze - mamy bardzo mało danych i trening będzie silnie utrudniony. Przykłady pojawiające się w tej instrukcji korzystały z kilkuset ilustracji zawierających ciasto marchewkowe (ma charakterystyczną fakturę i paletę barw, często występuje z kremem).

4. Przygotujmy startowy zbiór. Wczytajmy go z dysku, przeskalujmy zawarte w nim obrazy do rozmiaru 64x64, znormalizujmy kanały tak, by wartości były z zakresu 0 do 1, upewnijmy się, że zawiera odpowiednie elementy poprzez wyplotowanie kilku z nich. [🖼️]

5. Mamy bardzo mało danych w stosunku do złożoności stojącego przed nami zadania. Potrzebujemy więc skorzystać z każdej możliwej metody, która choć trochę zwiększy liczbę dostępnych przykładów - nawet sztucznie. W tym przypadku użyjemy tzw. *data augmentation*.
1. Wykorzystywany od początku przedmiotu framework powinien oferować funkcje wspierające ten typ wstępnego przetwarzania obrazów.
  2. Zmodyfikujemy więc użytkowany zbiór danych tak, by w każdej iteracji fotografie były w losowy sposób zmienione w stosunku do swojej oryginalnej formy (nie zrobimy tego zbyt agresywnie - bezpieczne transformacje to odbicie lustrzane, niewielki zoom, rotacja o bardzo mały kąt). [🖼️]
  1. Sprawdźmy, czy zastosowane przez nas *data augmentation* działa poprawnie - ograniczymy zbiór danych do kilku elementów i wielokrotnie je wyplotujemy (uwzględniając transformacje). Jeżeli wszystko zostało rozstawione prawidłowo, to poszczególne wystąpienia danego obrazu powinny się między sobą nieznacznie różnić (ten sam obraz powinien być czasem przekrywiony, czasem lekko przybliżony, czasem odbity względem osi symetrii). [🖼️]

## Work it harder, make it better

Teraz pora na wytrenowanie obu sieci i znalezienie ich właściwym parametrów (tradycyjnie - z użyciem metod gradientowych). Tym razem będzie jednak nieco trudniej - nasz cel optymalizacji jest bardzo nietypowy, a do tego zmienia się w czasie. Procedura uczenia będzie więc wymagać naprzemiennego korygowania wag - raz w generatorze, raz w dyskryminatorze.

6. Ze względu na wielokrokowy i niestandardowy proces uczący będziemy zmuszeni ręcznie wskazywać jakie składowe gradientu wymagają obliczenia oraz które wagi mają zostać zaktualizowane.
1. Zanim przystąpisz do właściwej implementacji tej sekcji, zrób prosty "rozruch". Upewnij się, że w wybranej bibliotece potrafisz zaimplementować następujące kroki. [🖼️]
  1. Przygotuj mały model składający się z dwóch warstw gęsto połączonych (może przyjmować 2-3 wartości na wejściu i tyleż samo zwracać, nie musi robić nic co miałoby jakiś sens - to tylko mikro-przykład na rozruch i lepsze zapoznanie się z biblioteką).
  2. Podaj niewielki *batch* losowych wartości na wejście takiego modelu i policz dowolną różniczkowalną funkcję z wyjścia, symulującą funkcję straty (*loss*) - to może być coś tak prostego jak "różnica między średnim wyjściem z modelu, a liczbą 42".
  3. Policz wartości pochodnych cząstkowych z tej funkcji - ale tylko po parametrach pierwszej z dwóch warstw! Framework powinien umożliwiać zrobienie tej operacji w przystępny sposób.
  4. Użyj tych pochodnych by skorygować wagi w tej właśnie warstwie - skorzystaj z dowolnego optymalizatora (np. SGD). Ponownie - powinno być to wspierane przez używaną bibliotekę.
  5. Wykonaj kilka iteracji, powtarzając powyższe kroki (nie losuj ponownie danych wejściowych). Czy wartości naszego psuedo-*loss* spadają?
7. Zaczniemy od łatwiejszej fazy - uczenia dyskryminatora. [🖼️]
1. Pociągnijmy ze zbioru danych *batch* obrazów - może być bardzo mały, np. 8 elementów (ponieważ cel optymalizacji jest trudny, to będziemy wykorzystywać dużo bardzo drobnych aktualizacji wag sieci). Te obrazy będą reprezentować klasę "prawdziwa fotografia", oznaczoną etykietą 1.
  2. Wygenerujemy drugie tyle (czyli np. też 8) sztucznych obrazów z użyciem generatora. Stworzymy *batch* składający się z 8 wektorów losowego szumu o rozkładzie normalnym (parametry takie jak kilka punktów wcześniej) i rozmiarach właściwych dla generatora.

Podajmy go na jego wejście i zgarnijmy wyniki. Pamiętajmy, by za każdym razem były to nowe, świeżo tworzone obrazy! One z kolei będą reprezentować klasę 0 - obrazy "fałszywe".

3. Sklejmy obie paczki obrazów w połączony *batch* o podwójnym rozmiarze (np.  $2 * 8 = 16$  obserwacji). Stwórzmy też odpowiedni wektor wynikowych etykiet.
  4. Zaciemnijmy nieco wektor etykiet, dodając do niego losowe wartości z zakresu  $\pm 0.05$ . Ta mała sztuczka pozwoli nam na mniej agresywną pracę dyskryminatora i potencjalnie lepszy przepływ gradientu.
  5. Taki *batch* to po prostu zestaw przykładów uczących do treningu dyskryminatora - po połowie z każdej klasy.
  6. Podaj *batch* na wejście dyskryminatora, pobierz zwrócone wyniki. Oceń ich jakość korzystając ze standardowej funkcji *loss* do takich celów - *binary cross entropy*. Zapisz gdzieś uzyskany wynik.
  7. Policz pochodne z *loss* po wagach dyskryminatora.
  8. Użyj je, by zaktualizować wagi w dyskryminatorze. Możesz skorzystać z najlepszych dostępnych w danej bibliotece narzędzi - np. optymalizatora *Adam*. Pamiętaj by uczenie przebiegało bardzo, bardzo, bardzo powoli - sprawdzają się tu bardzo niskie *learning rate*, wynoszące nawet 0.00001 (to nie jest literówka!).
  9. Pamiętaj, by wagi generatora nie uległy żadnym zmianom w tym kroku!
  10. Po tym kroku dyskryminator powinien odrobinę lepiej rozpoznawać fałszywki - choć zmiana ta może nie być bezpośrednio zauważalna.
8. Teraz generator musi nadrobić straty i zmierzać w kierunku tworzenia lepszych fotografii.




1. Przygotujmy nowy *batch* 8 wygenerowanych obrazów.
  2. Przygotujmy pasujący do niego wektor oczekiwanych etykiet - ale teraz chcemy, by fałszywe przykłady były zaklasyfikowane jako prawdziwe! Podajemy więc odwrotne etykiety, niż w poprzednim etapie. Tym razem nie dodajemy do nich żadnego szumu. Nie prezentujemy też dyskryminatorowi żadnych prawdziwych obrazów.
  3. Podajmy wygenerowane obrazy na wejście dyskryminatora, pobierzmy wynik, w oparciu o niego policzmy funkcję straty (pamiętajmy, że tym razem będziemy tym bardziej zadowoleni, im bardziej dyskryminator będzie uważał, że sztuczne obrazy były prawdziwymi). Zapiszmy uzyskany *loss*.
  4. Policzmy pochodne z *loss* po wagach generatora. Użyjmy ich do aktualizacji tychże wag (tak samo jak w poprzedniej fazie - można nawet skorzystać z tego samego optymalizatora).
  5. Tym razem musimy dopilnować, by zmieniały się tylko wagi generatora - dyskryminator pozostaje bez zmian!
  6. Po tym kroku generator powinien być w stanie tworzyć odrobinę lepsze fałszywki.
9. Trening tego typu modeli może trwać bardzo długo (prezentowane na koniec instrukcji przykłady zbiegły się po około 3000 epok). Na tak długi trening musimy się uprzednio przygotować.
1. Zaimplementuj pętlę, która powtarza kroki z punktu 7. i 8. kolejno dla wszystkich przykładów ze zbioru treningowego - w ten sposób uzyskamy implementację jednej epoki. Jeżeli chcesz, to skorzystaj w tym celu z narzędzi oferowanych przez preferowany framework.
  2. Przygotuj funkcję, która zapisuje aktualny stan obu modeli co N epok (np. dla  $N=10$  lub  $N=50$ ). Dzięki niej będzie istniała możliwość kontynuowania treningu po (spodziewanej lub nie) przerwie oraz powrotu do ostatniego stabilnego momentu, jeżeli procedura się wykończy (co zdarza się w przypadku rywalizujących sieci). Pamiętaj, by nie nadpisywać wcześniejszych wersji modelu (chcemy trzymać całą historię)!
  3. Przygotuj podobną funkcję, ale monitorującą efekty generacji.

1. Przygotuj i zapisz kilkanaście-kilkadziesiąt losowych wektorów, w formacie zgodnym z wejściem do generatora.
  2. Za każdym razem kiedy zapisujesz model, wygeneruj nowe obrazy na bazie tych właśnie wektorów (startowe wektory pozostają te same!).
  3. Obserwując jak ewoluowały w czasie powstające obrazy możemy wnioskować o procesie uczenia.
  4. Na koniec - postaraj się umożliwić również cykliczne zapisywanie informacji o tym, ile w danej epoce wynosił średni *loss* dla generatora, a ile dla dyskryminatora - powinny pozostawać we względnej równowadze!
  5. Sprawdź, czy twój preferowany framework oferuje mechanizmy wspierające taką archiwizację - większość z nich zapewnia wygodny sposób deklarowania *callbacków*, które mają się wykonać na koniec epoki.
10. Skoro wszystko już gotowe, to pora rozpocząć trening!
1. Na początku monitoruj efekty uczenia po każdej jednej epoce.
  2. Jeżeli jeden ze składowych modeli zbyt szybko zaczyna drastycznie wygrywać z drugim, to rozważ dodatkowe skrócenie kroku uczącego i/lub zmniejszenie zbyt silnego modelu (ograniczając jego ekspresywność i potencjał do *overfittingu*).
  3. W razie bardzo niepokojących wyników upewnij się, że w całą procedurę nie wkradły się błędy.
  4. Jeżeli wszystko pójdzie dobrze, to po przepracowaniu setek epok generator powinien zacząć wytwarzać obrazy przypominające te z oryginalnego zbioru.
    1. Nastaw się na ich przeciętną jakość - nasz dwusieczowy model jest dość prostym GANem (nie stosuje wielu nowoczesnych sztuczek i usprawnień) oraz pracuje na bardzo, bardzo małym zbiorze danych.
  5. Jak ("na oko") oceniasz rezultaty? Jak zmieniały się w czasie? [🖼️]

## Pożegnanie ze szkocką krata






Do tej pory stosowaliśmy generator uparty o konwolucje transponowane. Taki wariant ma nieco większą siłę wyrazu i łatwiej się go trenuje - ma jednak tendencje do generowania artefaktów w kształcie powtarzających się, siatkowatych tekstur. Alternatywą dla niego jest generator uparty o *upsampling* oraz zwykłe sieci konwolucyjne - i właśnie taki wariant zaimplementujemy w tej sekcji.

11. Przygotuj alternatywny generator. [🖼️]
1. Rozpocznij analogicznie jak w poprzednim podejściu - od wejściowego wektora losowych wartości i powiększenia go do tensora 8x8x128 z użyciem warstwy gęstej (ten fragment kodu można dosłownie skopiować).
  2. Następnie zastosujemy tzw. *upsampling blocks*.
    1. Każdy z nich zaczyna się od warstwy wykonującej zwiększenie rozdzielczości - czyli właśnie *upsampling* - w tym przypadku wystarczający będzie taki, który do interpolacji korzysta z najbliższych sąsiadów danego piksela.
    2. Po *upsamplingu* rozdzielczość kanałów powinna wzrosnąć dwukrotnie w każdym wymiarze przestrzennym.
    3. Następnie stosujemy następujące po sobie dwie (zwykle!) warstwy konwolucyjne - by uzupełnić świeżo powiększony obraz o brakujące szczegóły. Każda z nich składa się z:
      1. właściwej części konwolucyjnej (kernele 3x3),
      2. *batch normalisation*,

3. aktywacji *leaky ReLU* (nachylenie części negatywnej nadal 0.2).
3. Stosujemy kolejno trzy takie bloki. W pierwszym używamy 64 filtrów konwolucyjnych w każdej z wewnętrznych warstw, w drugim 128, a w trzecim 256. Wynikowy rozmiar tensora to 64x64x256.
4. Całość zamykamy w analogiczny sposób jak poprzednio - konwolucją o kernelu 5x5 i aktywacji typu *sigmoid* - sprowadzając wszystko do 3 kanałów RGB.
12. Przeprowadź trening z użyciem nowego generatora. Jak różnią się uzyskane wyniki? Jak tym razem przebiegała ich ewolucja? 

## No i co taki generator właściwie potrafi, he?

To już ostatnia sekcja zadań do wykonania! Na pożegnanie z sieciami generatywnymi wykonamy kilka ćwiczeń, których celem będzie przyjrzenie się bliżej temu, jaki potencjał ma wytrenowany generator oraz jak rozumieć przestrzeń losowych wejściowych cech, na której bazuje.

13. Spróbujmy tak dobrać wejściowe wartości do generatora, by odtworzyć jakiś rzeczywisty obraz. 
  1. Przyjrzymy się historii treningu. Zdecydujemy, po której epoce zwracane rezultaty wydają się najlepsze - i wczytajmy archiwalne wersje generatora pochodzące z tego właśnie momentu.
  2. Wybierzmy jeden obraz ze zbioru treningowego - najlepiej możliwie typowego przedstawiciela danej klasy.
  3. Stwórzmy losowy wektor wejściowy do generatora i wytwórzmy nową sztuczną fotografię.
  4. Niech naszą funkcją straty będzie różnica między wytworzoną fałszywą obserwacją, a wybranym wcześniej obrazem - dobrze zadziała tu błąd średniokwadratowy (MSE).
  5. Policzmy pochodne z tej straty po...poszczególnych cechach wejściowego wektora!
  6. Użyjmy optymalizatora gradientowego, by zaktualizować te cechy i zmniejszyć błąd (w tym wypadku wystarczy SGD z *learning rate* równym 0.01 i *momentum* równym 0.9).
    1. Wagi sieci nie ulegają żadnym zmianom! Na podstawie gradientu zmieniamy same dane wejściowe!
  7. Zapętlmy kilkanaście-kilkadziesiąt kroków takiej optymalizacji - powinno wystarczyć, by proces się ustabilizował.
  8. Zapiszmy ewolucję wynikowego obrazu - od losowego do takiego, który próbował imitować zadane wejście. 
  9. Zapiszmy też finalny wektor wejściowych wartości.
14. Pozmieniamy ręcznie pojedyncze wartości w zapisanym wcześniej wektorze. Jak zmiany te wpływają na wynikowy obraz? 
15. Spróbujmy powtórzyć eksperyment z punktu 13. - ale tym razem niech wzorcowy obraz będzie zupełnie niepowiązany z wytrenowaną klasą. Jak tym razem poradzi sobie generator? 
16. Wybierzmy inny obraz ze zbioru treningowego i dla niego również znajdziemy wektor wejściowy zwracający podobny rezultat (analogicznie jak w punkcie 13.).
  1. Wygenerujmy ciąg kilku-kilkunastu wektorów w taki sposób, że pierwszy zawiera wartości uzyskane dla obrazu A, ostatni wartości uzyskane dla obrazu B, a wszystkie pozostałe to interpolowane wartości pośrednie.
  2. Każdy z pośrednich wektorów podaj do generatora i wytwórz na jego bazie obraz.
  3. Jak wygląda ciąg takich "pośrednich" obrazów między A i B? 

17. Uwaga! Wymienione w tej sekcji kroki powtórzmy dla obu generatorów! Czy wyniki różnią się w zależności od użytego modelu? [📷]

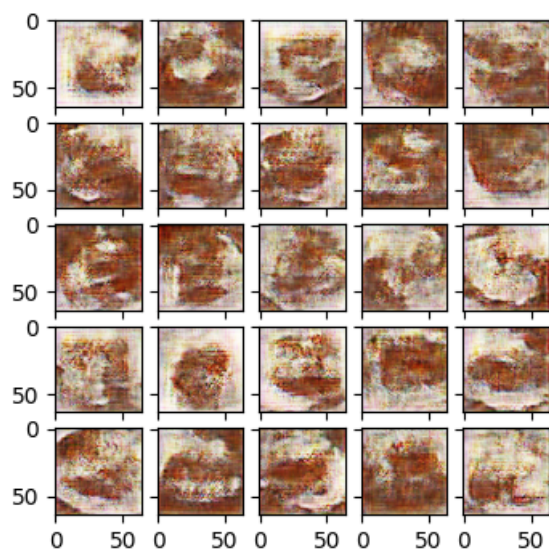
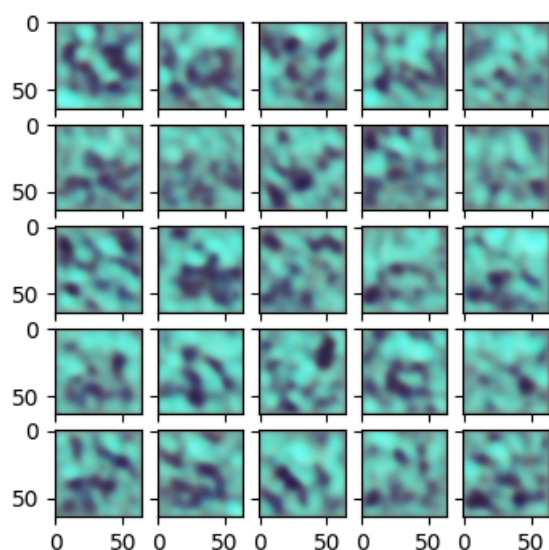
## Jesteśmy wolni - możemy iść!

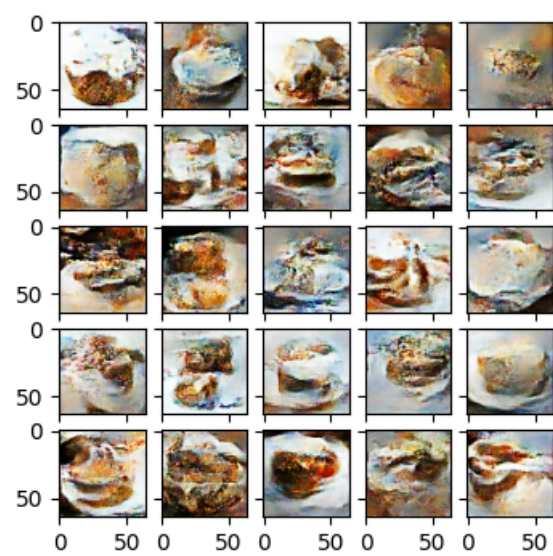
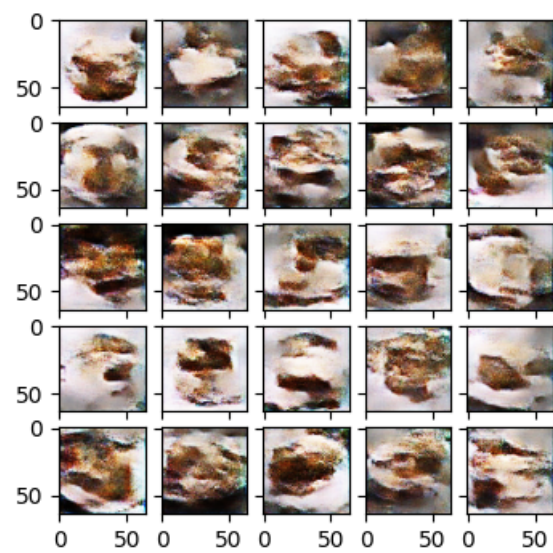
18. Zanim wyślesz finalny raport z tego (długiego!) zadania - spisz swoje ogólne wrażenia i refleksje z jego wykonania. Co było trudnym elementem implementacji? Co trwało najdłużej? Jakie nasuwają się końcowe wnioski?

## A jak mam takie wyniki, to jest dobrze, czy źle?

Konkretne efekty zależą od jakości zbioru danych, detali w implementacji architektury i...zwykłego szczęścia. Poniższe przykłady mają na celu przede wszystkim ustalenie rozsądnego punktu odniesienia - co do możliwej do uzyskania jakości i tego, jak zależy ona od momentu uczenia.

- Generowane obrazy dla 10, 300, 900 i 2000 epok (wariant z interpolacją).





- Oryginalny obraz oraz jego odtworzenie z wykorzystaniem modelu generatywnego.



