Dokumentacja końcowa

Zespół nr **16**: Jakub Polak, Hanna Smach

Zadanie: Generowanie obrazów samochodów przy użyciu sieci GAN. Do zaliczenia wystarczy wariant bezwzględny, na maksymalną ocenę parametryzowane np. poprzez typ auta.

1. Wstęp

W ramach projektu zaimplementowano 2 sieci – generatora i dyskryminatora, składające się na architekturę GAN - Generative Adversarial Network. Plik główny zapisano w formacie Jupyter Notebook. Dodatkowo załączono pliki *README* oraz *requirements.txt* (venv), które składają się na opis utworzenia wirtualnego środowiska oraz instalacji wymaganych bibliotek. Projekt był testowany na zasobach z dostępnością GPU-CUDA.

2. Instrukcja uruchomienia skryptu wraz z opisem implementacji sieci

Po utworzeniu wirtualnego środowiska zgodnie z krokami wskazanymi w README, należy uruchomić skrypt GAN_wersja_nieparam.ipynb (wersja nieparametryzowana) lub GAN_wersja_param.ipynb (wersja parametryzowana). Kolejne komórki pliku odpowiadają kolejnym etapom projektu.

Po zaimportowaniu bibliotek i utworzeniu roboczych folderów, gdzie przechowywane będą dane, skrypt ściąga plik .rar ze wskazanego URL i go rozpakowuje. Kolejny krok to segregacja zdjęć do odpowiednich katalogów potrzebnych do realizacji ewentualnego wariantu z parametryzacją. Definiowane są również typy tensorów Long i Float dla CUDA, którego wykorzystanie mocy obliczeniowej znacznie poprawia wydajność działania skryptu.

Ważnym etapem jest inicjalizacja parametrów, gdzie definiujemy liczbę klas równą **8** (tyle, ile typów aut – w wersji parametryzowanej), liczbę epok równą **10** w wersji nieparametryzowanej lub **20** w wersji parametryzowanej, żądany rozmiar zdjęć (**64x64** lub **256x256**), rozmiar pakietu zdjęć (z ang. batch) równy **128** oraz rozmiar wektora wejściowego dla generatora również równy **128**.

Z zapisanych zdjęć chcielibyśmy uzyskać tensory, dlatego po ustandaryzowaniu rozmiaru funkcją *resize()* (ponieważ w dataset obrazy mają różniące się wymiary) wykonujemy również transformację obrazów na PyTorchowe wektory i normalizację. Potem grupujemy w pakiety i tasujemy, tym samym otrzymując właściwy zbiór treningowy z pomocą *DataLoader()*.

Kolejny krok to przygotowanie klas generatora i dyskryminatora, które dziedziczą po klasie *Module* z biblioteki torch.nn. Każda z klas tworzy instancję z atrybutem modelu, gdzie definiowane są warstwy konwolucyjne sieci przy pomocy kontenera *Sequential*. Każda z klas posiada również funkcję *forward*, która wektor wartości, podany jako argument wejściowy, przekazuje do modelu, tzn. przekazuje na warstwy sieci i zwraca nowy wektor po przejściu przez wszystkie warstwy.

Dla klasy dyskryminatora warstwy tworzone są jako *Conv2d*, a dla generatora *ConvTranspose2d* (o różnicy tej wspomniano w dokumentacji wstępnej). Wykonywana jest również normalizacja danych wejściowych do warstwy (*BatchNorm2d*) w celu przyspieszania działania sieci i poprawy jej stabilności. Jako funkcję aktywacji pomiędzy warstwami wybrano LeakyReLu, która dopuszcza wartości mniejsze od zera. Natomiast jako ostatnią funkcję aktywacji dla dyskryminatora wybrano funkcję sigmoidalną, a dla generatora tangens hiperboliczny.

Generator w wersji sparametryzowanej od niesparametryzowanej różni się tym, że jako funkcję aktywacji wykorzystuje ReLU oraz przed warstwami konwolucyjnymi wejście sieci konkatenowane jest z zakodowaną klasą, którą chcemy uzyskać i przepuszczane przez warstwę gęstą.

Dyskryminator, podobnie do generatora wzbogacony został o warstwy gęste. W tym przypadku jednak jest ich kilka, następują po przejściu obrazu wejściowego przez warstwy konwolucyjne i skonkatenowaniu zakodowanej klasy zdjęcia do wyniku. Pomiędzy warstwami gęstymi jako funkcję aktywacji również wykorzystano LeakyReLU. Pomiędzy warstwami stosowany jest też dropout, który losowo wyłącza wagi neuronów w każdej iteracji.

Jeszcze przed treningiem sieci jako funkcję straty przyjmujemy binarną entropię krzyżową. Inicjalizujemy obiekty klasy dyskryminatora i generatora. Do optymalizacji procesu uczenia metodą spadku gradientu został wykorzystany algorytm Adam ze współczynnikiem uczenia równym 0.0002, co dawało najlepsze rezultaty. Zostały również zdefiniowane funkcje: do zapisywania obrazów wygenerowanych przez klasę generatora, do wyliczania metryki FID i osobno metryki IS. Funkcje wyliczania obu metryk zostały zaimplementowane w oparciu o wskazane źródła i model sieci Inception V3.

Proces uczenia sieci trwa 10 lub 20 epok w zależności od wersji. W każdej iteracji obejmuje zdefiniowanie tensorów dla dyskryminatora, które wskazują, że zdjęcie jest prawdziwe (tensor wypełniony 1) lub fałszywe (tensor wypełniony 0). Potem tworzony jest wektor szumu, podawany na wejście generatora, który zwraca obrazy, podawane kolejno na wejście dyskryminatora. Proces nauki generatora polega próbie „oszukania" dyskryminatora, tak by stwierdził, że są to zdjęcia prawdziwe. Mierzy to funkcja straty. Następnie trenuje się dyskryminator. Całkowity wynik funkcji straty dla dyskryminatora zawiera składowe od zdjęć realnych pochodzących ze zbioru treningowego, jak i tych fałszywych zwróconych przez generator. Co 100 iteracji zapisywane jest zbiór zdjęć generowanych przez generator, tak by móc obserwować skutek jego nauki.

Po procesie uczenia rysowane są wykresy funkcji straty dla generatora i dyskryminatora w zależności od liczby iteracji. Skrypt ma również możliwość zapisania modelu i jego załadowania, co jest krokiem opcjonalnym.

Ostatecznie, następuje ewaluacja sieci GAN. Pierwszym etapem jest wygenerowanie zestawu zdjęć z losowego szumu przez wytrenowany model generatora. Potem, po przygotowaniu modelu sieci Inception V3, zdefiniowane wcześniej funkcje wyliczają metryki FID oraz IS (ich znaczenie opisano w dokumentacji wstępnej).

3. Zmiany względem dokumentacji wstępnej

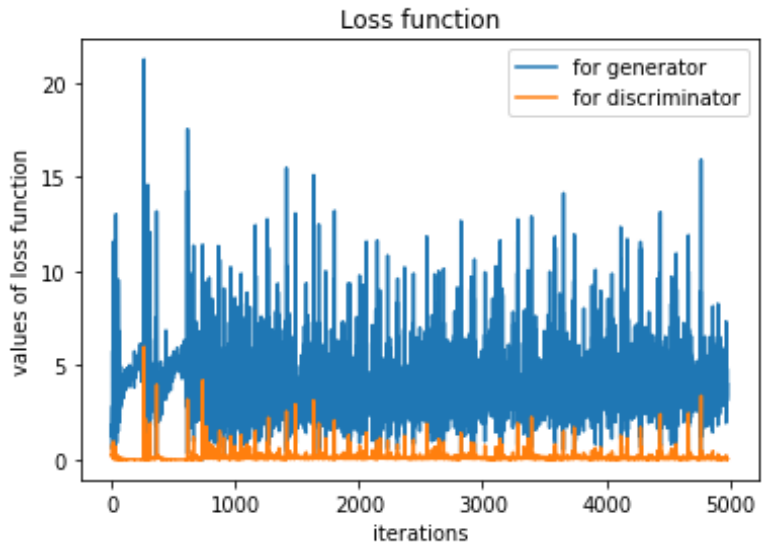
By zaprezentować wszystkie rezultaty projektu, zamiast jednego skryptu załączono 2 skrypty – osobne implementacje wersji nieparametryzowanej i parametryzowanej, różniące się przede wszystkim klasami dyskryminatora i generatora oraz procesem uczenia. Dodatkowo, w wyniku trudności

związanej z implementacją wersji parametryzowanej projektu, eksperyment zbadania, czy przy parametryzowaniu mniej liczne kategorie będą prawidłowo generowane, musiał zostać pominięty, gdyż nie udało się osiągnąć rozróżnialności klas.

4. Eksperymenty i wyniki – wersja nieparametryzowana

Jednym z eksperymentów była zmiana rozmiaru jądra (z ang. kernel) w warstwach konwolucyjnych sieci. Przetestowano 3 różne rozmiary. Jednocześnie generowano macierz obrazów o rozmiarze NxN, gdzie N=5 (zatem zestaw objął 25 obrazów). Wyniki przedstawiono poniżej:

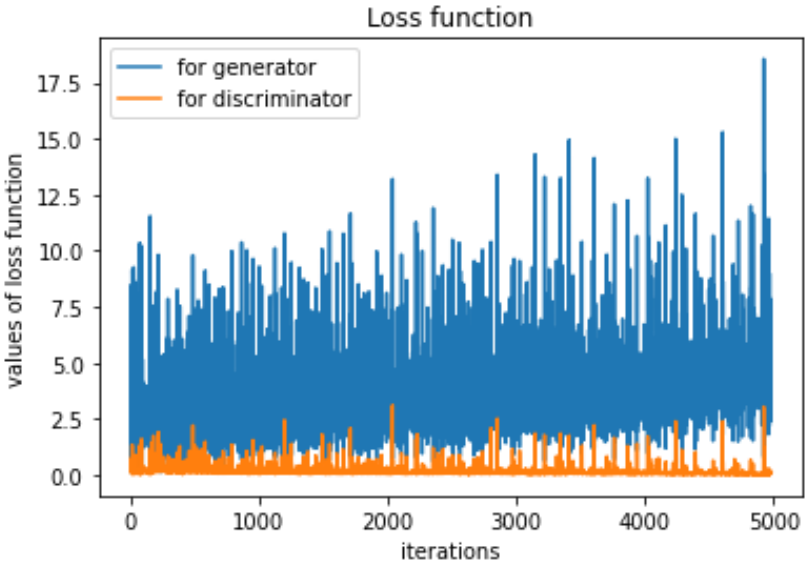
Rozmiar jądra: **4**
Liczba epok: 10
Liczba pakietów: 498
FID: 153,289
IS: 2,855
Funkcja straty dla generatora: krzywa waha się w okolicach wartości 4
Funkcja straty dla dyskryminatora: osiągane wartości poniżej 0,1



Rys.1. Funkcja straty w zależności od iteracji

Rys.2. Zestaw 25 obrazów wygenerowanych przez klasę generatora podczas ostatniej iteracji

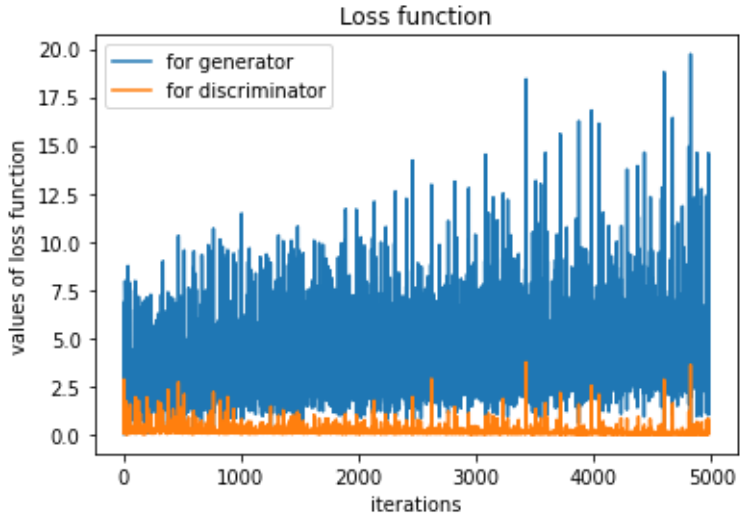
Rozmiar jądra: **6**
Liczba epok: 10
Liczba pakietów: 498
FID: 148,915
IS: 2,942
Funkcja straty dla generatora: krzywa waha się w okolicach wartości 5; ma charakter rosnący, a nie malejący
Funkcja straty dla dyskryminatora: osiągane wartości na poziomie ok. 0,1



Rys.3. Funkcja straty w zależności od iteracji

Rys.4. Zestaw 25 obrazów wygenerowanych przez klasę generatora podczas ostatniej iteracji

Rozmiar jądra: **8**
Liczba epok: 10
Liczba pakietów: 498
FID: 152,953
IS: 2,413
Funkcja straty dla generatora: bardzo duże oscylacje, trend rosnący krzywej
Funkcja straty dla dyskryminatora: osiągane wartości na poziomie ok. 0,1



Rys.5. Funkcja straty w zależności od iteracji

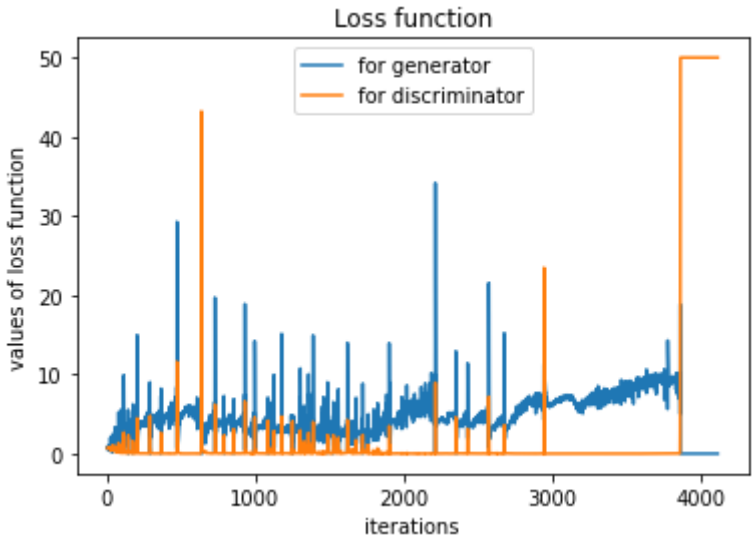
Rys.6. Zestaw 25 obrazów wygenerowanych przez klasę generatora podczas ostatniej iteracji

Można zauważyć, że wraz ze zwiększaniem rozmiaru jądra (kernel) funkcja straty dla generatora osiąga coraz większe niepożądane oscylacje. Jednakże nie musi występować taka zależność, gdy choć trochę zmienimy inne parametry sieci – inna kombinacja wraz ze zwiększeniem rozmiaru jądra może przynieść lepsze rezultaty. Jednak po wielu testach na zaimplementowanej przez nas sieci – rozmiar jądra równy 4 wraz z innymi hiperparametrami zwracał najlepsze wyniki.

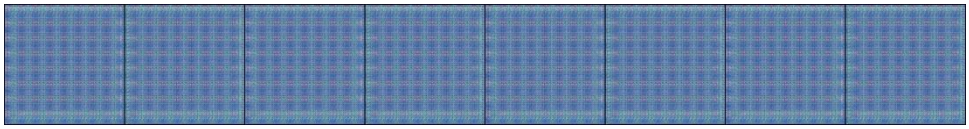
Ponadto, zwiększanie rozmiaru jądra wydłuża się również obliczenia – proces uczenia zajmuje więcej czasu. Zmiana rozmiaru nie wpłynęła znacznie na wartości metryki Inception Score (im wyższa wartość, tym lepiej), choć w przypadku zmiany rozmiaru z 4 na 6 spadla wartość metryki *Fréchet Inception Distance* (im niższa, tym lepiej). Być może w przypadku dalszego rozwoju projektu można byłoby uzyskać lepsze wyniki przy rozmiarze jądra warstwy konwolucyjnej równym 6 i dopasowaniu innych parametrów.

5. Eksperymenty i wyniki – wersja parametryzowana

Rozmiar jądra: 4
Liczba epok: 9
Liczba pakietów: 498
FID: 109,365
IS: 1.00



Rys.7. Funkcja straty w zależności od iteracji



Rys.8. Zestaw 8 obrazów reprezentujących 8 typ aut wygenerowanych przez klasę generatora

Jak łatwo zauważyć, wersja sparametryzowana ma bardzo duże problemy z uczeniem. Funkcja straty dla dyskryminatora ma charakter skokowy, po czym zatrzymuje się na dużej wartości, funkcja straty dla generatora ma również lekki charakter rosnący od ok. 2000 iteracji. Nie udało się poprawić procesu uczenia dla wersji parametryzowanej, zatem 8 obrazów reprezentujących klasy przedstawia tylko szum.

6. Komentarze i podsumowanie

Podczas trenowania sieci w celu uzyskania jak najlepszych wyników testowaliśmy i dobieraliśmy różne wartości hiperparametrów oraz różne kombinacje warstw, co obejmowało:

- zmianę współczynnika uczenia dla algorytmu Adam (różne dla generatora i dyskryminatora),
- różne wartości (nachylenia fragmentu krzywej) dla funkcji aktywacji LeakyReLu,
- zmiana funkcji LeakyReLu na ReLu,
- zastosowanie warstw konwolucyjnych i liniowych w różnych ilościach, wielkościach i kombinacjach zarówno w dyskryminatorze jak i generatorze,
- zmianę funkcji straty na błąd średniokwadratowy,
- label smoothing – zmianę wartości oczekiwanych dla zdjęć prawdziwych z 1.0 na 0.9, w celu zmniejszenia pewności dyskryminatora.

Analizując uzyskane wyniki i obrazy, można stwierdzić, że wersja nieparametryzowana radzi sobie lepiej niż parametryzowana, jednakże wersja parametryzowana było o wiele trudniejsza do implementacji. Obie wersje mają jednak potencjał do dalszego rozwoju.

Przy ewaluacji sieci wartości IS oraz FID nie powinno się porównywać między wersjami, jedynie osobno dla każdych z wersji w przypadku zmiany parametrów, gdyż są to metryki względne. W przypadku wersji nieparametryzowanej wartości IS są na niskimi poziomie (jest to cecha metryki), natomiast powinny być na dużo wyższym poziomie dla opcji z klasami, czego nie udało się jeszcze uzyskać w tym projekcie.

7. Bibliografia

1. https://medium.com/swlh/gan-to-generate-images-of-cars-5f706ca88da
2. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
3. https://learnopencv.com/conditional-gan-cgan-in-pytorch-and-tensorflow/#CGAN_PyTorch
4. https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/cgan/cgan.py
5. https://machinelearningmastery.com/how-to-evaluate-generative-adversarial-networks/
6. https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/
7. https://github.com/mseitzer/pytorch-fid
8. https://medium.com/octavian-ai/a-simple-explanation-of-the-inception-score-372dff6a8c7a
9. https://github.com/sbarratt/inception-score-pytorch