GSN - Final Checkpoint

Paweł Przybyłowski 131813
Informatyka, specjalizacja ISWD
stopień 2, semestr 2
pawel.przybylowski@student.put.poznan.pl

Marianna Murawicz 132291
Informatyka, specjalizacja ISWD
stopień 2, semestr 2
marianna.murawicz@student.put.poznan.pl

I. Opis problemu badawczego

Problem badawczy wybrany przez naszą grupę będzie dotyczył siódmego tematu z listy, tj. Style transfer. Tematem będzie translacja zdjęć rzeczywiste - gra komputerowa oraz zmiana barw obrazu wyjściowego. Na wejściu podawane będą 2 obrazy - jeden z fotografią, której styl ma być zmieniony oraz drugi z docelowymi barwami, jakie ma przyjąć obraz wynikowy. Zdecydowaliśmy się przeprowadzić taką translację, ponieważ uważamy, że niektóre gry komputerowe wyróżniają się oryginalnym stylem graficznym i ciekawe będzie zobaczenie nowych krajobrazów. Prace opierać będziemy na zrzutach ekranu z gry „Borderlands", gdzie przykładowy został przedstawiony na rysunku 1. Obraz z barwami należy rozumieć jako zdjęcie zrobione np. podczas danej pory roku (w zime przeważająca ilość zimnych kolorów, śnieg i analogiczne w inne).

Rysunek 1. Zrzut ekranu z gry „Borderlands"

II. Zaplanowane końcowe wymagania projektu

- Stworzenie zbioru danych – w przypadku gry – pobranie z dostępnych źródeł oraz ręczne tworzenie zrzutów ekranu, w przypadku krajobrazów – pobranie z dostępnych źródeł.
- Zaprojektowanie i implementacja modelu sieci CycleGAN, która na wejściu przyjmuje 2 zdjęcia.

III. Realizacja końcowych wymagań projektu

A. Zbiór danych

Ostateczny zbiór danych zawiera 388 zrzutów ekranu z gry. W grze występują 4 różne rejony, które różnią się od siebie kolorystyką (zimowy [Rys. 2], pustynny [Rys. 3], roślinny [Rys. 4] i zwyczajny [Rys. 5], który niczym się nie wyróżnia). Starano się wykonać około tyle samo zrzutów z każdego z nich. Zbiór danych jest nieco mniejszy niż początkowo planowano, ponieważ sam proces tworzenia był dość żmudny – do gry Borderlands 2 nie ma żadnej modyfikacji, która umożliwia latanie. Zostały jedynie znalezione modyfikacje wysokości skoku za pomocą programu CheatEngine. Pozwoliły one na wykonanie zdjęć krajobrazów z lotu, ale pojawił się problem animacji ruchu dłoni podczas skoku (tylko podczas pojedynczych klatek obie dłonie się „chowają", więc trzeba było dobrze wybrać moment na zrzut ekranu). Część dotycząca realnych krajobrazów została pobrana ze zbioru danych udostępnionego przez autorów artykułu [1] i znajduje się w nim 6853 zdjęć.

Rysunek 2. Zimowy zrzut ekranu

Rysunek 3. Pustynny zrzut ekranu

Rysunek 4. Roślinny zrzut ekranu

Rysunek 5. Zwyczajny zrzut ekranu

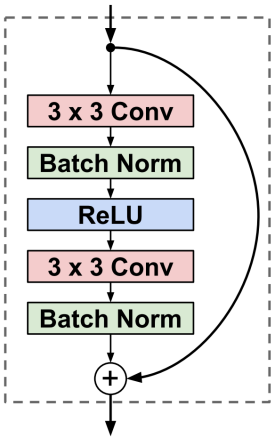
B. Budowa modelu

Stworzone zostały proste modele sieci generatora i dyskryminatora (klasy z prefiksem „Simple"), na których trening modelu okazał się być nieefektywny i porzucono dalsze prace w ich kierunku. Kolejnym etapem była budowa modelu, który został zaproponowany przez autorów pracy [1]. Odtworzyliśmy go na tyle ile opisane było w samym artykule.

Podstawowa architektura sieci generatora została przedstawiona w tabeli I, gdzie została ona przez nas zmodyfikowana. Struktura residual block została przedstawiona na rysunku 6.

Layer	Activation size
Input	3 x 256 x 256
32 x 9 x 9 conv, stride 1	32 x 256 x 256
64 x 3 x 3 conv, stride 2	64 x 128 x 128
128 x 3 x 3 conv, stride 2	128 x 64 x 64
Residual block, 128 filters	128 x 64 x 64
Residual block, 128 filters	128 x 64 x 64
Residual block, 128 filters	128 x 64 x 64
Residual block, 128 filters	128 x 64 x 64
Residual block, 128 filters	128 x 64 x 64
64 x 3 x 3 conv, stride 1/2	64 x 128 x 128
32 x 3 x 3 conv, stride 1/2	32 x 256 x 256
3 x 9 x 9 conv, stride 1	3 x 256 x 256

Tablica I
Podstawowa architektura dla sieci transferu stylu [2]



Rysunek 6. Struktura residual block [2]

Architektura sieci dyskryminatora została przedstawiona w tabeli II.

Layer
conv-LeakyReLU, 64 filters, stride 2
conv-norm-LeakyReLU, 128 filters, stride 2
conv-norm-LeakyReLU, 256 filters, stride 2
conv-norm-LeakyReLU, 512 filters, stride 2
conv, 1 filter, stride 1

Tablica II
Architektura sieci dyskryminatora

IV. Trening modelu i wyniki

Sieć została trenowana przez 200 epok, początkowo z wejściem w postaci 1 zdjęcia, aby sprawdzić czy model w ogóle ma prawo działać. Sieć odpalono z następującymi parametrami:

- learning rate - 0.0002
- decay_epoch - 100
- lambda - 10

Początkowo trening prowadzono na CPU i w tym przypadku rozdzielczość zdjęć została bardzo pomniejszona (64x64). Autorzy artykułu własny model trenowali na zdjęciach rozdzieczości 256x256, co ukazuje różnicę w mocy obliczeniowej maszyn.

Niestety z tego etapu prac nie ma żadnych wyników, ponieważ szybko przeszliśmy na obliczenia na GPU.

Dostępna maszyna miała w karcie graficznej dostępne 8GB pamięci, co pozwoliło na powiększenie rozdzielczości zdjęć (1920x1080 → 128x128) w porównaniu do obliczeń na CPU. Samo powiększenie rozdzielczości pozytywnie wpłynęło na otrzymywane wyniki już po kilku pierwszych epokach (mierzone empirycznie).

Na rysunkach 7, 8, 9 oraz 10 przedstawiono wyniki działania sieci (input - output).

Rysunek 7. Wynik działania sieci

Rysunek 8. Wynik działania sieci

Rysunek 9. Wynik działania sieci

Rysunek 10. Wynik działania sieci

Wyniki okazały się nie być idealne – wiele brakowało do stylu, który próbowaliśmy uzyskać, ale przede wszystkim nie były widoczne grube kontury, które są charakterystyczne dla Borderlands. Można odnieść wrażenie, że model zmieniał głównie barwy zdjęć na bardziej stonowane i zimne, co w pewnym sensie jest elementem stylu tej gry.

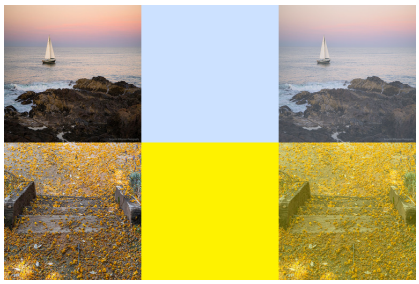
Mimo wszystko przeszliśmy do dalszych prac, czyli implementacji, która obsługuje dwa zdjęcia wejściowe. Stwierdziliśmy, że rozwiązanie dotyczące konkatenacji kanałów zdjęć [3] będzie najprostsze w implementacji, a skoro zostało polecone przez autora, to bardzo możliwe że poprawnie zadziała.

Zdjęcia wczytywane są za pomocą klasy DataLoader dostępnej w torchu. Na tym etapie już pojawił się problem, ponieważ pracowaliśmy na systemie Windows, gdzie parametr num_workers powodował błędy (dlatego ostatecznie został ustawiony na 0).

Utworzone zostały 3 DataLoadery do wczytywania zdjęć. Następnie na początku każdej epoki zdjęcia treningowe ze zbioru A były konkatenowane za pomocą funkcji torch.cat na 3 „wymiarze” – zdjęcia [W, H, C] zostały konkatenowane do [W, H, 2C]. W związku z tą zmianą należało zmienić wejściową ilość kanałów dla modelów generatora i dyskryminatora. Nie znaleźliśmy rozwiązania jak zdjęcie 6-kanałowe poprawnie przetransformować spowrotem w 3-kanałowe (RGB), dlatego uznaliśmy, że przeanalizujemy i przetestujemy kod autorów artykułu, z nadzieją że zaimplementowali taki mechanizm.

Okazało się, że autorzy nie zawarli tego w swojej pracy, dlatego szukaliśmy innego rozwiązania. W między czasie uruchomiliśmy trening na oryginalnym kodzie i pomimo tego, że zmuszeni zostaliśmy do kolejnego zmniejszenia wczytywanych zdjęć (64x64), wyniki wydały się bardzo obiecujące, dlatego stwierdziliśmy, że zbadamy jego działanie.

Wczytywanie dwóch zdjęć zrealizowaliśmy za pomocą mieszania (blend). Wykorzystana została funkcja Image.blend z pakietu Pillow. Oba zdjęcia początkowo zostawały konwertowane z RGB do RGBA, a następnie łączone (na podstawowe zdjęcie nakładane było drugie z kanałem alpha=0.3). Na samym końcu zdjęcie zostawało spowrotem konwertowane do RGB, aby miało 3 kanały. Dzięki temu obeszliśmy problem nadmiarowej ilości kanałów. Działanie łączenia zdjęć zostało przedstawione na rysunku 11.



Rysunek 11. Przykładowe łączenie zdjęć

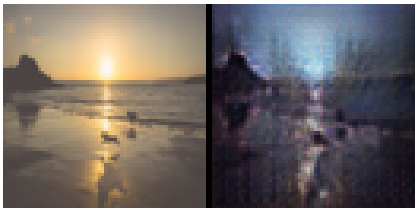
Wykorzystanie oryginalnego kodu wymagało również stworzenie własnego DataLoadera, który wczytywałby dodatkowe, trzecie, zdjęcie, a następnie je mieszał (kod zamieszczony jest w pliku „custom_dataset” w głównym folderze repozytorium).

Opisana metoda łączenia zdjęć zadziałałaby również na naszym modelu, ale chcieliśmy zbadać o ile lepiej zrobi to poprawny model – nasz z pewnością nie był napisany w 100% poprawnie.

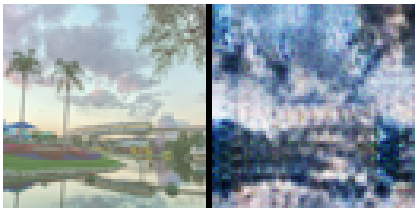
Zdjęcia z drugiego zbioru treningowego (nazwany trainA2) były wygenerowane i przedstawiały jeden kolor. Przykładowe zostały przedstawione na rysunku 12.

Rysunek 12. Przykładowe zdjęcia ze zbioru trainA2

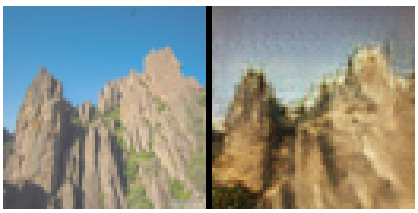
Na rysunkach 13, 14, 15, 16 i 17 zostały przedstawione wyniki działania tej sieci. Zdjęcie wejściowe zostało przedstawione już po sklejeniu z racji, że DataLoader od razu je łączył i takie przekazywał do sieci. Mimo wszystko można jednak zauważyć, że wejściowe zdjęcie na rysunku 16 jest nienaturalnie żółte – prawdopodobnie połączone zostało z żółtym odcieniem.



Rysunek 13. Wynik działania oryginalnej sieci



Rysunek 14. Wynik działania oryginalnej sieci



Rysunek 15. Wynik działania oryginalnej sieci

Rysunek 16. Wynik działania oryginalnej sieci

Rysunek 17. Wynik działania oryginalnej sieci

Wyniki ponownie nie były idealne, ale tym razem można zauważyć, że zdjęcia wyjściowe miały już bardziej wyrazisty kontur, zbliżony do tego w Borderlands. Widać to szczególnie na rysunkach 15, 16, 17. Zdjęcia nadal miało dużo szumy, co znacznie pogarszało ich jakość, ale otrzymane efekty były zadowalające.

Warto zaznaczyć, że model ten trenował się nieprzerwanie około 2 dni, a skończone zostało tylko 8 epok. Jest to wyjątkowo ciekawe, że po tak krótkim treningu otrzymano tak dobre wyniki i jednocześnie o wiele lepsze niż za pomocą naszego modelu.

Na ostatniej stronie raportu dołączony został wykres funkcji straty w czasie tego modelu. Można z niego zauważyć, że dość szybko wartość funkcji straty spadła do około 3, a w kolejnych epokach stabilizowała się i nieco spadała. Można zatem założyć, że jeśli trening trwałby zamierzone 200 epok (co niestety było dla nas niemożliwe), wyniki byłyby o wiele lepsze.

V. Podsumowanie

Stworzony przez nas model okazał się działać, ale jego wyniki nie były zadowalające, pomimo że w pewnym stopniu nauczył się stylu gry. Bardzo możliwe, że stworzona sieć mogła zostać udoskonalona, a także sam proces trenowania. Oprócz tego warto byłoby uruchomić trening na większej liczbie epok, aby zweryfikować jego działanie.

Model zaproponowany przez autorów oryginalnego rozwiązania już po kilku epokach prezentował bardzo ciekawe, a zarazem poprawne wyniki. Również w tym wypadku warto byłoby przeprowadzić trening na większej ilości epok, aby osiągnąć zamierzony cel.

Kolejnym aspektem, który pozytywnie wpłynąłby na jakość wyników, jest rozdzielczość zdjęć wejściowych. Zmuszeni byliśmy skalować je do rozdzielczości 128x128, a nawet 64x64, gdzie ilość szczegółów jest znacznie mniejsza i unikalny styl samych zdjęć na tym cierpi.

Ostatecznie łączenie zdjęć przeprowadzone było w inny sposób niż początkowo zakładaliśmy. Nie znaleźliśmy rozwiązania, dzięki któremu konkatenacja zdjęć na kanałach działała tak, abyśmy uzyskali zdjęcie wyjściowe 3-kanałowe. Stworzono bardzo prosty mechanizm łączenia zdjęć za pomocą mieszania ich na kanale przezroczystości.

Samo zagadnienie transferu stylu jest bardzo ciekawe i z pewnością znajdzie swoje zastosowanie. Już teraz można wskazać jak oryginalnym i efektownym rozwiązaniem jest stosowanie takich sieci w grach. Przykładem może być

zmiana stylu w czasie rzeczywistym zaprezentowana przez Unity [4].

VI. Kod źródłowy

Kod źródłowy znajduje się na publicznym repozytorium na GitHubie pod linkiem https://github.com/Stfoorca/GSN˙StyleTransfer.

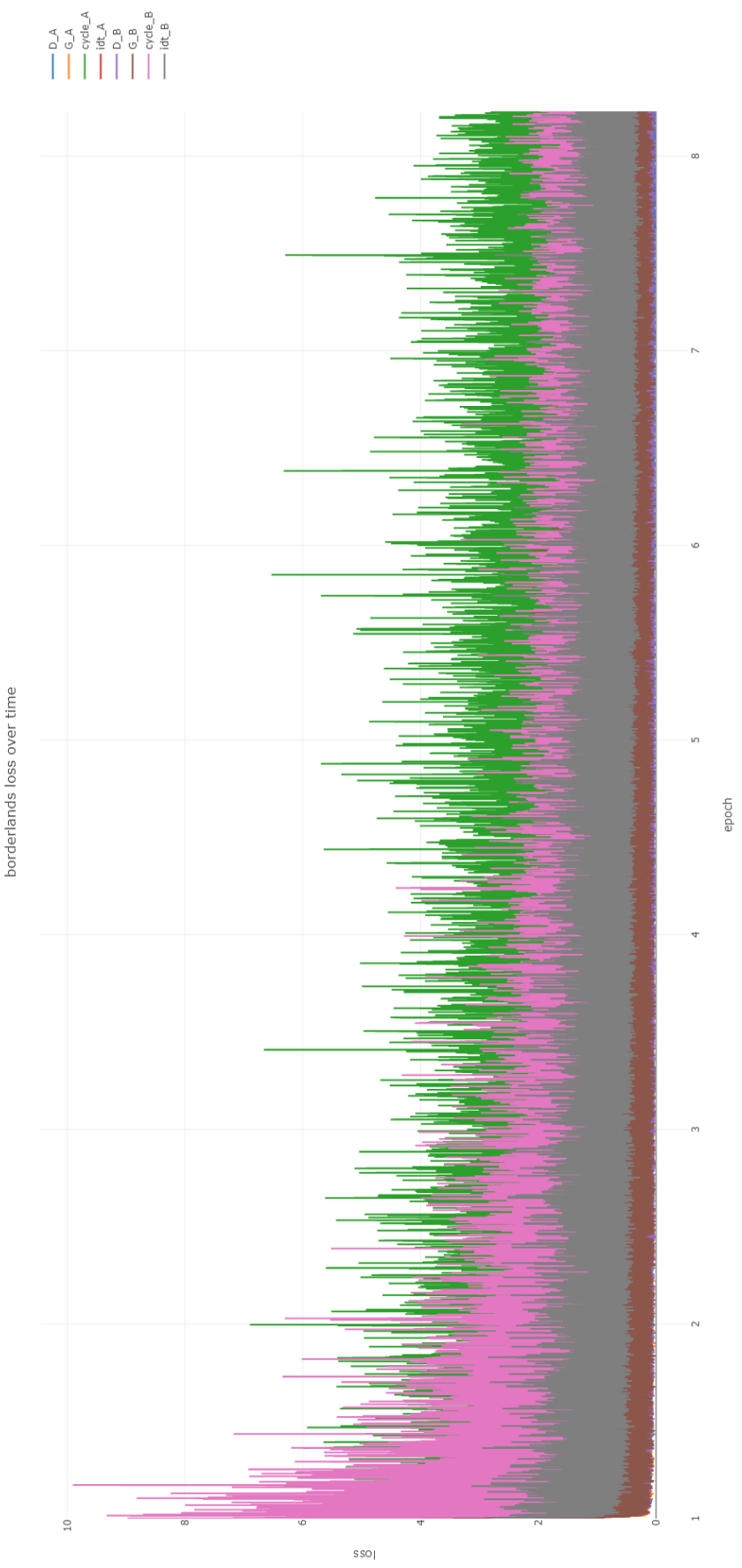
Literatura

[1] Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros, Translation using Cycle-Consistent Adversarial Networks, [Online] Available: https://arxiv.org/pdf/1703.10593.pdf

[2] Justin Johnson, Alexandre Alahi, Li Fei-Fei, Perceptual Losses for Real-Time Style Transfer and Super-Resolution - Supplementary Material, [Online] Available: https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf

[3] Jun-Yan Zhu, Taesung Park, Tongzhou Wang, Implementation of CycleGAN, [Online] Available: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/498

[4] Thomas Deliot, Florent Guinier and Kenneth Vanhoey, Real-time style transfer in Unity using deep neural networks, [Online] Available: https://blogs.unity3d.com/2020/11/25/real-time-style-transfer-in-unity-using-deep-neural-networks/



Załącznik 1. Wykres funkcji straty w czasie