Image Completion Network

Hubert Skrzypczak - 172063
Adam Stańlej - 171794
Andrzej Świerczyński - 171573

Założenia projektu

Naszym celem było stworzenie sieci neuronowej, która pobiera obraz z maską od użytkownika, a następnie generuje realistyczne wypełnienie dla wyciętego obszaru. Nasze rozwiązanie opiera się o dwie sieci: generator i dyskryminator. Generator odpowiada za tworzenie ww. obrazów, natomiast dyskryminator ocenia, cze dany obraz jest wygenerowany, czy prawdziwy. Generator był trenowany na zdjęciach plaż i wybrzeży, więc na nich radzi sobie najlepiej.

Budowa sieci

Generator

Wejście

Macierz (256 x 256 x 3)

Warstwa 1.

- konwolucja (filters = 64, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)
- max pooling

Warstwa 2.

- konwolucja (filters = 2*64, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)
- max pooling

Warstwa 3.

- konwolucja (filters = 4*64, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)
- max pooling

Warstwa 4.

- konwolucja (filters = 8*64, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)
- max pooling

Warstwa 5.

- konwolucja (filters = 16*64, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)
- max pooling

Warstwa 6.

- transponowana konwolucja (filters = 8*64, kernel_size=3, padding="same")
- konkatenacja z warstwą 4
- konwolucja (filters = 96, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)

Warstwa 7.

- transponowana konwolucja (filters = 4*64, kernel_size=3, padding="same")
- konkatenacja z (warstwą 3) * 0.8
- konwolucja (filters = 96, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)

Warstwa 8.

- transponowana konwolucja (filters = 2* 64, kernel_size=3, padding="same")
- konkatenacja z (warstwą 2) * 0.4
- konwolucja (filters = 96, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)

Warstwa 9.

- transponowana konwolucja (filters = 64, kernel_size=3, padding="same")
- konkatenacja z (warstwą 1) * 0.2
- konwolucja (filters = 96, kernel_size=3, padding="same")
- batchNormalization
- leaky ReLU (alpha=0.001)

Wyjście

konwolucja (filters = 3, kernel_size=1, activation=sigmoid)

Dyskryminator

Wejście

Macierz (256 x 256 x 3)

Warstwy konwolucyjne

z sieci VGG16 wytrenowanej na imagenet (zamrożone wagi)

Warstwy głębokie

Dwie warstwy gęste po 128 neuronów z funkcją aktywacyjną ReLU

Wyjście

Warstwa gęsta z 1 neuronem z funkcją aktywacyjną sigmoid

Loss generatora ma trzy składowe:

contextual_loss_valid - L1 liczone dla pixeli poza wycietymi fragmentami
contextual_loss_hole - L1 liczone dla pixeli z wycietych fragmentow
perceptual_loss - binary crossentropy z tego ile razy udało się oszukać dyskryminator

$\textbf{loss}_{\textbf{G}}$ = contextual_loss_valid + 6 * contextual_loss_hole + 0.1 * perceptual_loss

Loss dyskryminatora ma dwie składowe:

fake_loss - binary crossentropy z true negative
real_loss - binary crossentopy z true positive

$\textbf{loss}_{\textbf{D}}$ = fake_loss + real_loss

Wybranie architektury i hiperparametrów generatora odbywało się na zasadzie eksperymentów.

Trenowaliśmy różne architektury.

Pierwszym podejściem było, aby generator miał architekturę **autoencodera**, jednak szybko okazało się, że przy większej ilości warstw gradient zanikał, a proces uczenia ustawał.

Drugim architekturą wzorowana na architekturze **u-net**. Tutaj sytuacja wyglądała inaczej, gradient nie zanikał ze względu na dodatkowe połączenia pomiędzy warstwami, jednak dziura z pierwszego zdjęcia propagowała się do wyjścia.

Trzecim, tym który okazał się sukcesem, była również architektura **u-net**, ale z dodatkowymi współczynnikami, na dodatkowych połączeniach między warstwami.

Eksperymentalne dobieranie architektury było przeprowadzane na zbiorze CIFAR10 (mniejsze zdjęcia, dawały szybszy rezultat). Dopiero w kolejnym kroku najlepsze warianty trenowaliśmy na domyślnym zbiorze (zdjęcia plaż) i wybraliśmy, tą która w początkowych kilkudziesięciu epokach uczyła się najlepiej.

Uczenie odbywało się w dwóch etapach:

Etap 1.

Uczenie równocześnie dyskryminatora i generatora z hiperparametrami:

- batch_size = 8
- gen_lr = 1e-4
- dis_lr = 1e-7
- image_in_epoch = 8 * 320

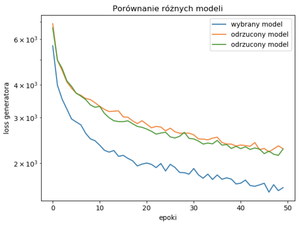
W obu przypadkach używany był optymalizator Adam

Learning rate dyskryminatora jest niższy, ponieważ uczymy go tak zwanym transfer learningiem, jego warstwy konwolucyjne są już wytrenowane, tutaj dotrenowujemy tylko 3 ostatnie warstwy gęste. Umożliwia to dostosowanie wcześniej wytrenowanej sieci do naszych potrzeb i skraca czas potrzebny na uczenie.

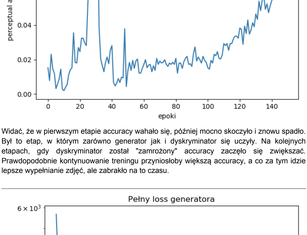
Etap 2.

Dyskryminator jest już wytrenowany i jego skuteczność jest bliska 100%.
W tym etapie uczymy tylko generator z takimi samymi parametrami.

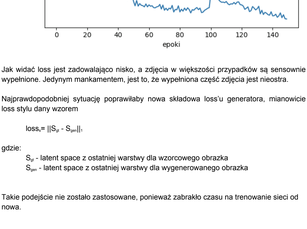
Uczenie przebiegało następująco:



Widać wyraźną przewagę ostatecznie wybranego modelu nad wcześniej testowanymi rozwiązaniami.



Widać, że w pierwszym etapie accuracy wahało się, później mocno skoczyło i znowu spadło. Był to etap, w którym zarówno generator jak i dyskryminator się uczyły. Na kolejnych etapach, gdy dyskryminator został "zamrożony" accuracy zaczęło się zwiększać. Prawdopodobnie kontynuowanie treningu przyniosłoby większą accuracy, a co za tym idzie lepsze wypełnianie zdjęć, ale zabrakło na to czasu.



Jak widać loss jest zadowalająco nisko, a zdjęcia w większości przypadków są sensownie wypełnione. Jedynym mankamentem, jest to, że wypełniona część zdjęcia jest nieostra.

Najprawdopodobniej sytuację poprawiłaby nowa składowa loss'u generatora, mianowicie loss stylu dany wzorem

$$\text{loss}_s = ||S_d - S_{we}||_f$$

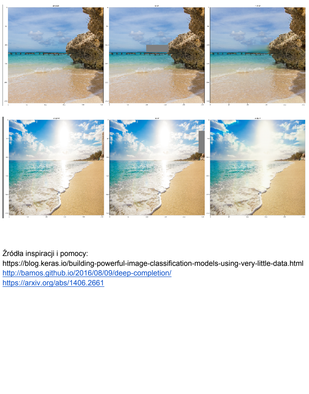
gdzie:

S_d - latent space z ostatniej warstwy dla wzorcowego obrazka
S_{we} - latent space z ostatniej warstwy dla wygenerowanego obrazka

Takie podejście nie zostało zastosowane, ponieważ zabrakło czasu na trenowanie sieci od nowa.

Przykładowe wypełnienia obrazów

Od lewej: zdjęcie oryginalne, zdjęcie z wyciętą dziurą, zdjęcie po wypełnieniu.



Źródła inspiracji i pomocy:
https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
http://bamos.github.io/2016/08/09/deep-completion/
https://arxiv.org/abs/1406.2661