Dokumentacja projektu

Neural Style Transfer

Adam Łaba
Aleksander Kluczka
Paweł Sipko
22 stycznia 2022

1. Wstęp

Jako temat projektu wybraliśmy odtworzenie algorytmu NST opisanego w pracy A Neural Algorithm of Artistic Style której autorami są: Leon A. Gatys, Alexander S. Ecker oraz Matthias Bethge. Działanie tego algorytmu polega na przeniesienie *stylu* z jednego obrazu do *zawartości* drugiego, w ten sposób tworząc nowy obraz wynikowy.

Rys1. Przedstawienie działania algorytmu

Jak można zauważyć na powyższej grafice, NST pozwala na utworzenie zupełnie nowego obrazu wykorzystując *styl* i *zawartość* obrazów wejściowych.

2. Opis działania algorytmu

Niech :
S – obraz wejściowy z *stylem*,
Z – obraz wejściowy z *zawartością*,
W – obraz wynikowy
Wtedy algorytm można przedstawić za pomocą poniższego pseudokodu:

```
1  Inicjalizacja W
2  Pętla iteracyjna:
3     Obliczenie wartości funkcji strat w danej iteracji:
4         strata = różnica w stylu między S a W
5         strata += różnica w zawartości między Z a W
6         strata += total variation loss dla W
7     Obliczenie gradientu
8     Wykorzystanie optymalizatora do przekształcenia W
9  Koniec pętli
10 wynik = W
```

2.1. Inicjalizacja obrazu wynikowego

W naszej implementacji, podobnie jak w oryginalnej pracy obraz wynikowy powstaje poprzez zaszumienie obrazu wejściowego z *zawartością*.

2.2. Obliczenie wartości funkcji strat:

Pierwszym etapem jest wyodrębnienie zawartości i stylu z obrazów. Problem ten jest realizowany poprzez wykorzystanie wytrenowanej sieci neuronowej do klasyfikacji obrazów (w tym projekcie wykorzystaliśmy sieć VGG19, która została wytrenowana przy wykorzystaniu bazy obrazów imagenet).

2.2.1. Różnica *stylu*

Do wyodrębnienia *stylu* wykorzystywane są odpowiedzi neuronów poszczególnych warstw splotowych sieci (convolutional layers). Pojedyncza taka warstwa zawiera w sobie N różnych neuronów, a każdy z nich w odpowiedzi na obraz wejściowy generuje charakterystyczną dla siebie macierz (feature map).

Następnie, poprzez złączenie tych macierzy wyprodukowanych w obrębie jednej warstwy otrzymujemy macierz reprezentującą odpowiedzi wszystkich neuronów w danej warstwie. Posiadając taką macierz, obliczamy na jej podstawie macierz Grama:

$$G_{ij} = \sum_k F_{ik} F_{jk},$$

Wzór 1

Gdzie:
F – feature map dla konkretnej warstwy
W ten sposób otrzymujemy macierz reprezentującą *styl* obrazu.
Różnica między *stylem* dwóch obrazów (strata *stylu*) jest dla konkretnej warstwy w pierwotnym NST obliczana ze wzoru:

$$E = \frac{1}{4N^2M^2} \sum_{i,j} \left(G^W_{i,j} - G^S_{i,j} \right)^2,$$

Wzór 2

Gdzie:
N - ilość feature map w obrębie warstwy
M – rozmiar map (iloczyn ich szerokości i wysokości)
G^W – macierz Grama obrazu wynikowego
G^S – macierz grama obrazu ze *stylem*
W naszej implementacji pominęliśmy w tym wzorze wyraz N.

Na końcu sumujemy straty z obliczone dla każdej warstwy uwzględniając odpowiadające konkretnym warstwom wagi:

$$L_S = \sum \omega_l E_l,$$

Wzór 3

Gdzie:
ω_l - waga odpowiadająca warstwie l
E_l – strata odpowiadająca warstwie l

2.2.2. Różnica *zawartości*

W celu wyodrębnienia *zawartości* zaczynamy od utworzenia macierzy F będącej feature mapą dla konkretnej warstwy splotowej sieci (tak jak w poprzednim przypadku). Posiadając takie macierze dla obrazu wynikowego i obrazu wejściowego z *zawartością*, możemy w prosty sposób obliczyć stratę *zawartości*:

$$E = \frac{1}{M} \sum \left(F^W - F^Z\right)^2,$$

Wzór 4

Gdzie:
M - rozmiar map (iloczyn ich szerokości i wysokości)
F^W – feature map obrazu wynikowego
F^Z – feature map wejściowego obrazu z *zawartością*
W pierwotnym algorytmie NST suma była dzielona przez 2, natomiast w naszej implementacji dzielimy ją przez M.

Na końcu sumujemy straty z obliczone dla każdej warstwy uwzględniając odpowiadające konkretnym warstwom wagi.

$$L_Z = \sum \omega_l E_l,$$

Wzór 5

Gdzie:
ω_l - waga odpowiadająca warstwie l
E_l – strata odpowiadająca warstwie l

Warto dodać, że nie jest konieczne wykorzystywanie wszystkich warstw sieci – do obliczenia różnicy *zawartości* w zupełności wystarczy wykorzystanie jednej konkretnej warstwy, a do *stylu* kilku (najczęściej spotyka się z wykorzystaniem 5 warstw).

2.2.3. *Total variation loss*

Podczas tworzenia nowego obrazu wynikowego samoistnie pojawia się szum – widoczne są piksele o wartości skrajnie różnej od otoczenia. W celu minimalizacji tego zjawiska, w każdej iteracji do funkcji strat dodajemy wyraz związany z zaszumieniem obrazu – obliczamy sumę wartości bezwzględnej różnic między sąsiadującymi pikselami.

$$L_{TV} = \omega \sqrt{\sum_x ||\nabla I(x)||_2^2},$$

Wzór 6

Gdzie:
ω - waga TVL
I – obraz, dla którego obliczamy stratę

2.3. Obliczenie gradientu, przekształcenie *W*

Na podstawie funkcji strat:

$$L = L_S + L_Z + L_{TV},$$

Wzór 7

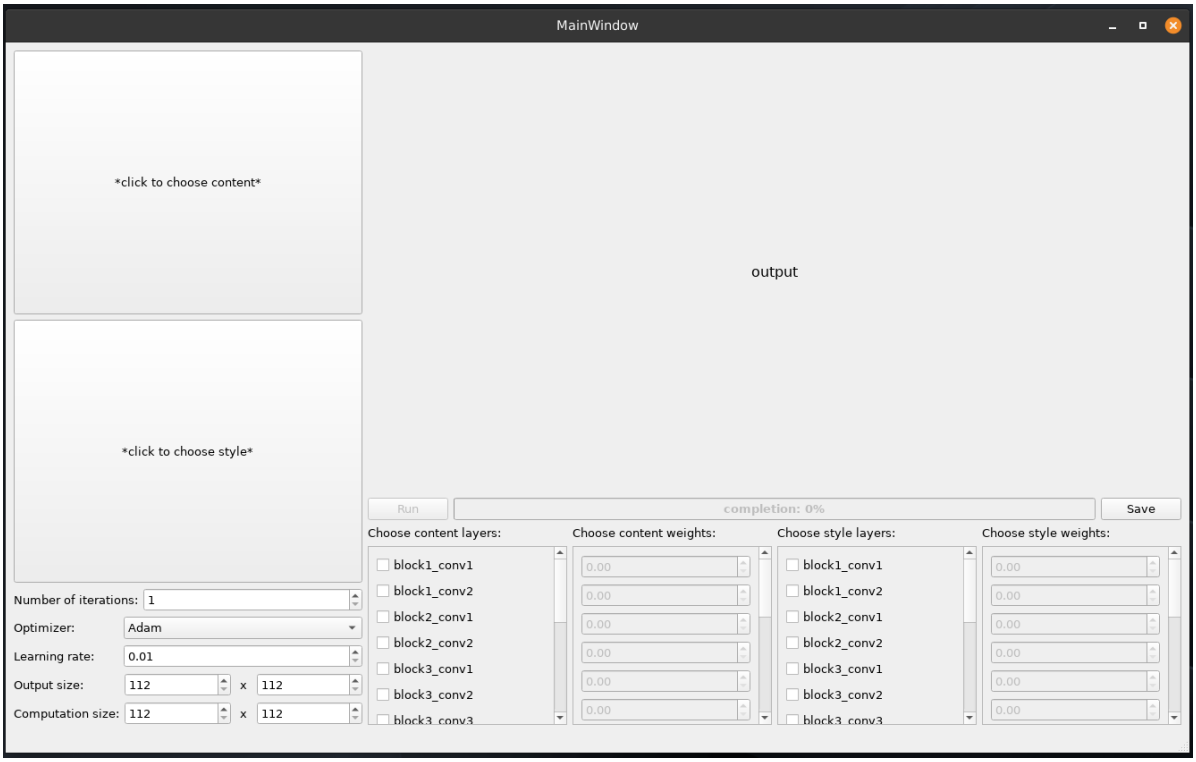
obliczamy komputerowo jej gradient względem pikseli obrazu wynikowego. Następnie, wykorzystujemy optymalizator, czyli funkcję służącą do minimalizacji błędu, która przekształca obraz wynikowy.

Całość powtarzamy iteracyjnie do momentu uzyskania satysfakcjonującego wyniku.

3. Interfejs użytkownika

Na pierwszy rzut oka interfejs użytkownika może być trudny do opanowania, jednak poniższy opis powinien rozjaśnić wszelkie wątpliwości.

Do czynienia mamy z poniższą aplikacją:



Rys2. Pusty interfejs

Na początek warto zwrócić uwagę, że do wykonania obliczeń potrzebujemy danych wejściowych, jak na przykład obraz content, obraz style, liczba iteracji, funkcja optymalizacji czy wagi poszczególnych obrazów. Jak widać na załączonym powyżej obrazie, przycisk “Run” jest na ten moment zablokowany, jako że użytkownik nie uzupełnił wszystkich wymaganych danych wejściowych.

Klikając na sporych rozmiarów przycisk po lewej stronie interfejsu otrzymujemy możliwość wybrania dowolnego obrazu z dysku. Wspierane formaty to JPG, PNG oraz BMP.

Po wybraniu obrazów content i style, należy wybrać oczekiwaną liczbę iteracji. Domyślnie ustawiona jest wartość 1, która powinna zostać zmieniona, jeśli chcemy uzyskać zadowalające rezultaty.

Zaraz poniżej można wybrać funkcje optymalizującą oraz współczynnik “learning rate”, który jest istotnym parametrem algorytmu NST. Sugerujemy nieznacznie zwiększyć jego wartość.

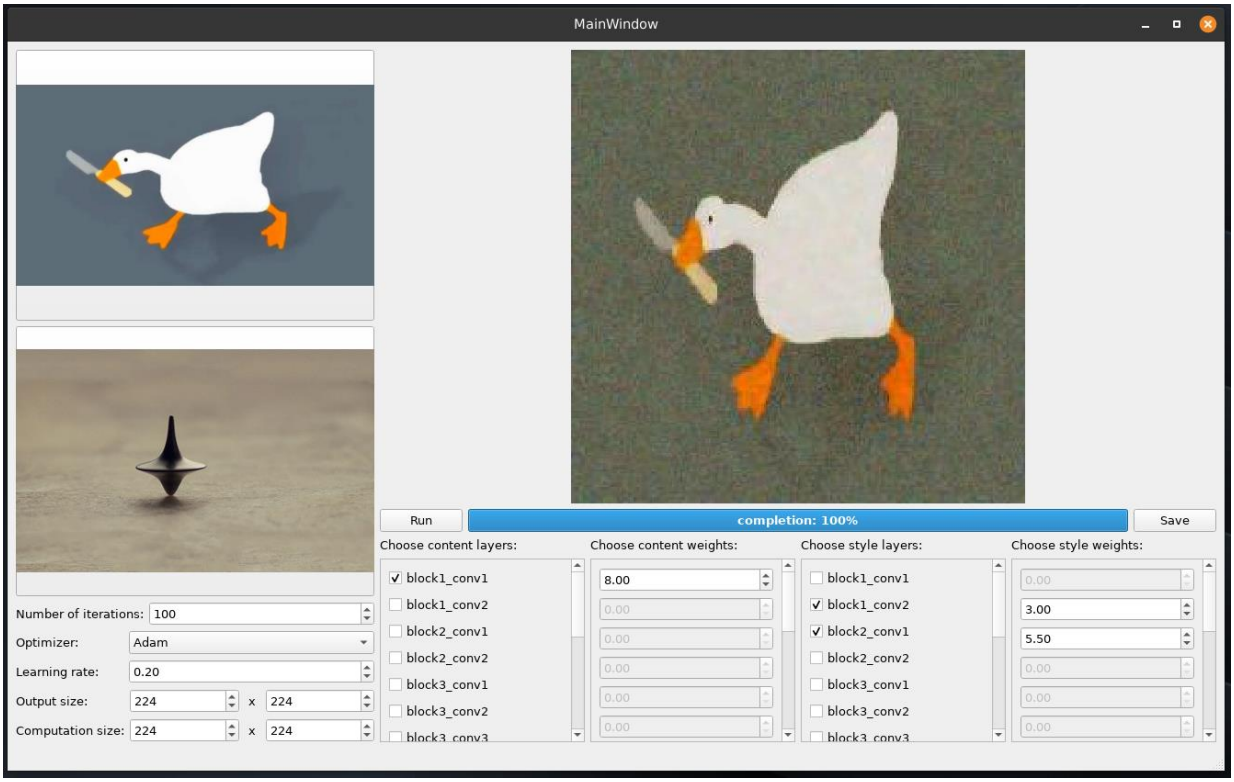
Na dole listy znajdują się pola do ingerencji w rozmiar wynikowego obrazu oraz wymiary obliczeń algorytmu.

W dolnym obszarze aplikacji znajdują się dwie przewijane listy, które pozwalają na wybranie warstw sieci neuronowej do odpowiednio zawartości oraz stylu obrazów. Po zaznaczeniu danej warstwy odblokowuje się możliwość ustawienia wagi tej warstwy w stosunku do pozostałych wag. Użytkownik musi wybrać conajmniej jedną warstwę dla zawartości oraz jedną dla stylu, a następnie ustawić ich wagi na wartości niezerowe (żeby dana warstwa faktycznie brałą udział w obliczeniach).

Na środku okna powinien odblokować się przycisk “Run”, po którego kliknięciu uruchomi się algorytm NST. Odblokuje się wtedy pasek postępu, który mniej więcej wskaże status ukończenia obliczeń.

Dodatkowo, przed kliknięciem “Run”, użytkownik ma możliwość wybrania ścieżki docelowej, do której wygenerowany zostanie obraz wynikowy. Domyślnie ustawiona jest lokalizacja pliku wykonywalnego tej aplikacji.

Po wypełnieniu wszystkich wymaganych pól oraz uruchomieniu algorytmu, aplikacja powinna wyglądać mnie więcej tak:



Rys3. Interfejs po wygenerowaniu obrazu

4. Testy

Testy rozpoczęliśmy od stworzenia bazowego, uniwersalnego zestawu danych wejściowych pozwalającego na uzyskanie zadowalających rezultatów, niezależnie od przekazanych obrazów content oraz style.

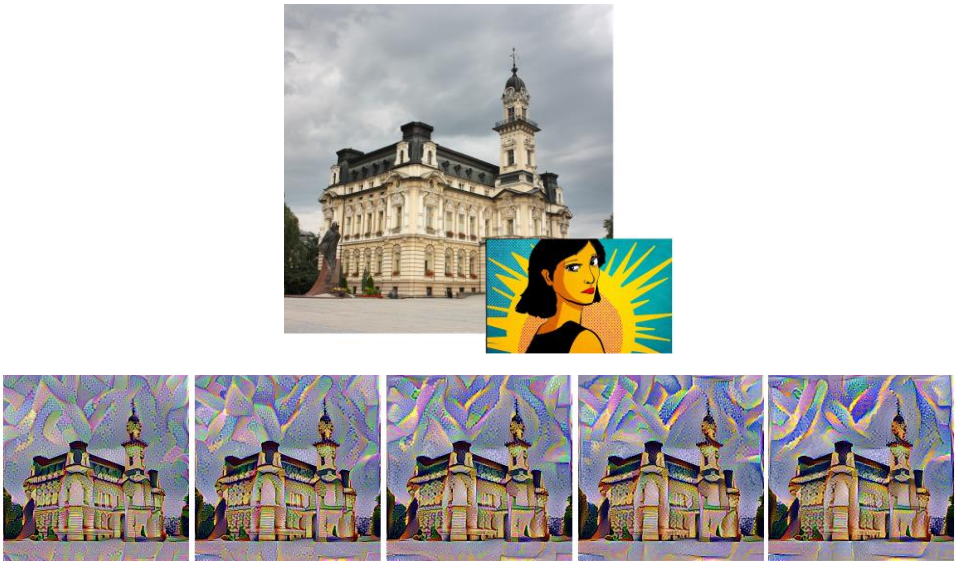
Skorzystaliśmy z pięciu warstw stylu oraz jednej warstwy zawartości z wagami odpowiednio: [1.0, 1.5, 2.0, 2.5, 3.0] i [10.0]. W celu odszumienia obrazu ustaliliśmy total_variation_loss_weight na 100.0. Wykorzystywanym przez nas optymalizatorem był Adam z danymi wejściowymi: learning_rate = 0.05, beta_1 = 0.99, epsilon = 0.1. Z uwagi na rozmiar obrazów wykorzystanych do trenowania sieci VGG19, przyjęliśmy rozmiar obliczonego obrazu

jak i obrazu wynikowego jako: 224 na 224. Ilością iteracji, która pozwoliła nam na uzyskanie zadowalających rezultatów w rozsądnym czasie była liczba 5000.

Mając tak przygotowany model zaczęliśmy badać wpływu pojedynczych danych wejściowych na otrzymywane rezultaty.

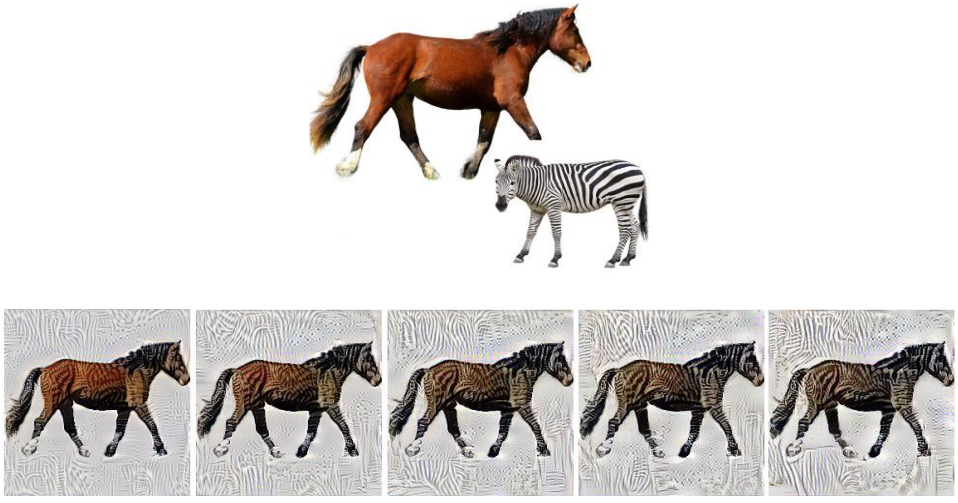
Na początku przeprowadziliśmy testy dla liczby iteracji. Poniższe przykłady zostały stworzone dla kolejno: 1000, 2000, 3000, 4000 i 5000 iteracji.

Pierwszy z nich zakłada zupełnie niepowiązane ze sobą obrazy style oraz content:



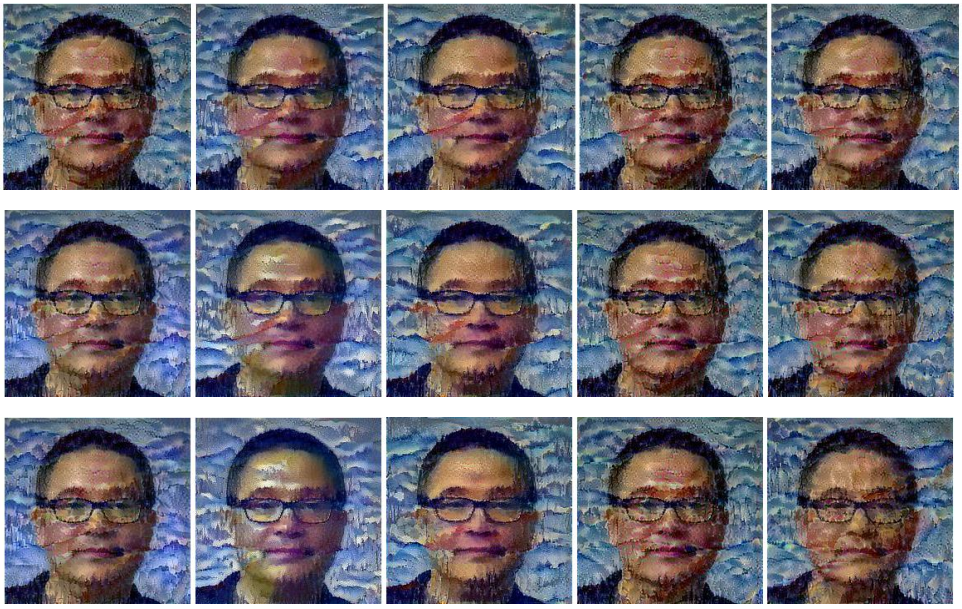
Rys4. Przykład zmiany rezultatu w zależności od liczby iteracji dla niepowiązanych obrazów.

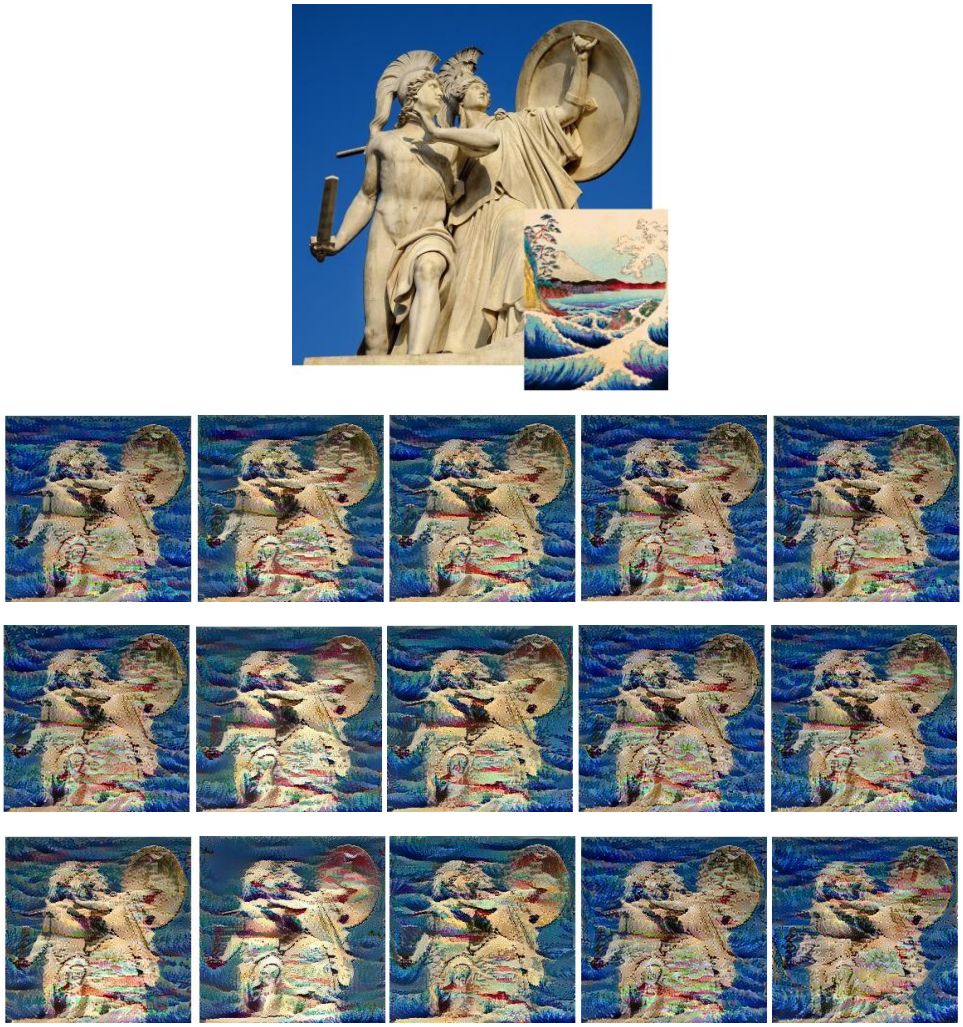
Drugi przykład opiera się na obrazach o podobnej zawartości i zróżnicowanych stylach:



Rys5. Przykład zmiany rezultatu w zależności od liczby iteracji dla obrazów powiązanych zawartością.

Znając wpływ iteracji na otrzymywane rezultaty, przeszliśmy do testowania wag poszczególnych warstw stylu. Jednocześnie utrzymywaliśmy stosunek sumy wag stylu do zawartości na poziomie kolejno: 1e2, 1e3, 1e4. W poniższych przykładach korzystaliśmy z dwóch warstw zawartości z niezmiennymi wagami [5.0, 5.0] oraz pięciu warstw stylu. Ponadto w celu uwydatnienia różnic zwiększyliśmy learning_rate do 0.2 i ograniczyliśmy liczbę iteracji do 3000.





Rys 6 i 7. Przykłady zmiany rezultatu w zależności od wag poszczególnych warstw stylu – poziomo - wyróżniamy kolejne warstwy, pionowo - zwiększamy stosunek sumy wag warstw stylu do sumy wag zawartości.

Na koniec zbadaliśmy wpływ wybranego optymalizatora. Do testowania wykorzystaliśmy dane z przygotowanego przez nas modelu i porównaliśmy użyty tam optymalizator Adam z innymi dostępnymi w bibliotece Keras - Adadelta i RMSprop.

Rys8. Przykład zmiany rezultatu w zależności od użytego optymalizatora – kolejno Adam, Adadelta i RMSprop.

5. Podział pracy

Pracę nad projektem rozdzieliliśmy wobec siebie:
Opracowanie algorytmu NST – Adam
Stworzenie GUI – Aleksander
Przygotowanie obrazów i testy - Paweł
Oczywiście jest to duże uproszczenie. W przypadku natrafienia na jakiś problem przerastający konkretną osobę staraliśmy się wspólnie szukać rozwiązania.

6. Źródła

Wykorzystane technologie:
Algorytm NST - tensorflow, keras, Pillow, numpy
GUI - PyQt5
Optymalizacja, liczenie na GPU - CUDA, cuDNN

Dodatkowo:
Przykładowa implementacja NST - keras
Przykładowa implementacja NST - tensorflow

7. Licencje

Większość zawartych w dokumentacji jak i w repozytorium przykładowych obrazów pochodzi z darmowych bibliotek ze zdjęciami.

- Nowy_Sacz_Ratusz.jpg
  Autor: Ffolas
  Licencja
- old_CGI.jpg
  Autor: Dk3dknight
  Licencja
- Woodcut.jpg
  Autor: Hiroshige Andō
  Licencja - domena publiczna z uwagi na wygasłe prawa autorskie