Sprawozdanie

Optymalizacja procesów dyskretnych

Wykonał: Julia Rojek (272529)

Semestr letni: 2024/25

Przedmiot: Optymalizacja procesów dyskretnych

Data zajęć: Poniedziałek

Godzina zajęć: 9.15-10.45

## Spis treści

[Spis treści 2](#_Toc198154463)

[1. Wstęp 3](#_Toc198154464)

[2. Przygotowanie danych 3](#_Toc198154465)

[3. Operacje im2col i col2im. 4](#_Toc198154466)

[4. Wktoryzowana konwolucja 4](#_Toc198154467)

[5. Transponowana konwolucja 5](#_Toc198154468)

[6. Funkcje aktywacji i metryki 6](#_Toc198154469)

[7. Architektura autoenkodera 6](#_Toc198154470)

[8. Procedura treningowa 7](#_Toc198154471)

**­­­­­Spis rysunków**

[Rysunek 1 Implementacja zadnia 1 4](#_Toc194760504)

[Rysunek 2 Implementacja algorytmu SortR 6](#_Toc194760505)

[Rysunek 3 Implementacja algorytmu Schrage 8](file:///C:\Users\julia\Desktop\greblicki%20sprawko1.docx#_Toc194760506)

# Wstęp

Celem projektu było zaimplementowanie od podstaw autoenkodera konwolucyjnego w języku Python z wykorzystaniem biblioteki NumPy. Zadaniem autoenkodera jest nauczenie się kompresji oraz rekonstrukcji obrazów wejściowych w skali szarości o rozmiarze 64×64 piksele.

W ramach realizacji projektu zaimplementowano wszystkie niezbędne elementy modelu, w tym:  
– operacje konwolucyjne w postaci wektoryzowanej (z użyciem im2col/col2im),  
– transponowaną konwolucję (tzw. dekonwolucję),  
– funkcje aktywacji (ReLU, Sigmoid) i ich pochodne,  
– funkcję straty (błąd średniokwadratowy – MSE),  
– prostą metrykę oceny jakości rekonstrukcji (binarna dokładność pikseli).

Projekt nie korzysta z gotowych frameworków do uczenia maszynowego (takich jak TensorFlow czy PyTorch), a wyniki działania zostały przeanalizowane oraz przedstawione graficznie.

# Przygotowanie danych

##### Opis zbiorów

Do uczenia autoenkodera wykorzystano własny zbiór obrazów zapisanych lokalnie w strukturze katalogów:

* train/ – obrazy treningowe,
* test/ – obrazy testowe (niewykorzystywane podczas uczenia, służące do ewaluacji jakości rekonstrukcji).

Obrazy znajdują się w formacie .jpg i zostały umieszczone w osobnych folderach w zależności od klasy (np. happy, banana). Na potrzeby tego projektu skupiono się na jednej wybranej klasie, aby uprościć problem i skoncentrować się na samej architekturze autoenkodera.

Każdy obraz został przeskalowany do rozmiaru 64×64 piksele i przekonwertowany do skali szarości (tryb L w bibliotece PIL), aby uprościć strukturę danych i zredukować liczbę kanałów do jednego. Dzięki temu każdy obraz reprezentowany jest jako macierz o wymiarach (64, 64).

##### **Funkcja wczytywania**

Do wczytywania danych zaimplementowano funkcję load\_images\_from\_folder(folder, max\_images=None). Jej zadaniem jest:

* przejście po wszystkich plikach .jpg w zadanym folderze,
* konwersja obrazu do skali szarości (jeśli jest w innym trybie),
* przeskalowanie obrazu do ustalonego rozmiaru (64×64),
* normalizacja wartości pikseli do zakresu [0, 1] poprzez podzielenie przez 255,
* opcjonalne zastosowanie dodatkowej transformacji (np. augmentacji), jeśli taka zostanie przekazana,
* zwrócenie zbioru obrazów w postaci tablicy NumPy o wymiarach (liczba\_obrazów, 64, 64) i typie float32.

Dodatkowo funkcja obsługuje parametr max\_images, który umożliwia ograniczenie liczby wczytywanych obrazów – przydatne w fazie testowania.

W przypadku braku plików .jpg w folderze, funkcja informuje o tym w konsoli.

# Operacje im2col i col2im.

Operacje im2col oraz col2im zostały zaimplementowane w celu umożliwienia wektoryzowanej realizacji konwolucji, co znacznie przyspiesza obliczenia.  
Zamiast wykonywać klasyczne przesuwanie filtra po obrazie w pętli, obrazy są przekształcane do postaci macierzy, w której każdy wiersz reprezentuje jeden fragment obrazu (tzw. „patch”), odpowiadający położeniu filtra.

* im2col(x, filter\_h, filter\_w, stride, pad)  
  Funkcja dzieli każdy obraz w zbiorze na fragmenty o rozmiarze filtra (filter\_h, filter\_w), przesuwane o zadany krok (stride) oraz z odpowiednim dopełnieniem (pad). Zwraca macierz, w której każdy wiersz odpowiada jednemu wycinkowi obrazu.
* col2im(col, x\_shape, filter\_h, filter\_w, stride, pad)  
  Funkcja odwrotna do im2col, która rekonstruuje oryginalne obrazy na podstawie kolumnowej reprezentacji oraz zadanych parametrów filtra, przesunięcia i dopełnienia. Operacja ta wykorzystywana jest m.in. w backpropagacji w celu obliczenia gradientów względem danych wejściowych.

# Wktoryzowana konwolucja

1. **Forward pass**

W celu przeprowadzenia operacji konwolucji w sposób wektoryzowany zaimplementowano funkcje:

* conv\_forward\_im2col – konwolucja dla pojedynczego filtra (np. w dekoderze),
* conv\_forward\_im2col\_multi – konwolucja z wieloma filtrami (np. w enkoderze).

Działanie:

1. Wejściowe obrazy są przekształcane do macierzy przy użyciu im2col,
2. Filtry są przekształcane do postaci kolumnowej,
3. Następnie wykonywane jest mnożenie macierzy wejściowej przez filtry,
4. Dodawane są biasy,
5. Wynik przekształcany jest z powrotem do postaci tensora o wymiarach  
   (N, F, H\_out, W\_out).
6. **Backward pass**

Do obliczenia gradientów względem wejścia, wag i biasów zaimplementowano:

* conv\_backward\_im2col – wersja dla pojedynczego filtra,
* conv\_backward\_im2col\_multi – wersja dla wielu filtrów (pełna konwolucja).

Działanie:

1. Gradient z wyjścia (dout) jest odpowiednio przekształcany,
2. Obliczany jest gradient względem wag przez przemnożenie transponowanej macierzy col z dout,
3. Obliczany jest gradient względem biasu przez sumowanie po wymiarze batcha,
4. Obliczany jest gradient względem wejścia przez przemnożenie dout z transponowanymi wagami, a następnie przekształcenie przez col2im.

# Transponowana konwolucja

1. Forward pass

Transponowana konwolucja, nazywana również dekonwolucją, została zaimplementowana w funkcji conv\_transpose\_forward\_im2col. Umożliwia ona rekonstrukcję obrazu z niższowymiarowej reprezentacji.

Działanie:

1. Wejściowe dane są najpierw odpowiednio „rozrzedzane” (upsampling), tj. wstawiane są zera pomiędzy wartościami, zależnie od stride,
2. Filtr zostaje odwrócony w pionie i poziomie,
3. Następnie wykonywana jest klasyczna konwolucja (forward) na danych rozszerzonych,
4. Otrzymany wynik jest przycinany lub dopełniany tak, aby pasował do oczekiwanego rozmiaru obrazu.
5. Backward pass

Funkcja conv\_transpose\_backward\_im2col odpowiada za propagację gradientu przez warstwę transponowanej konwolucji.

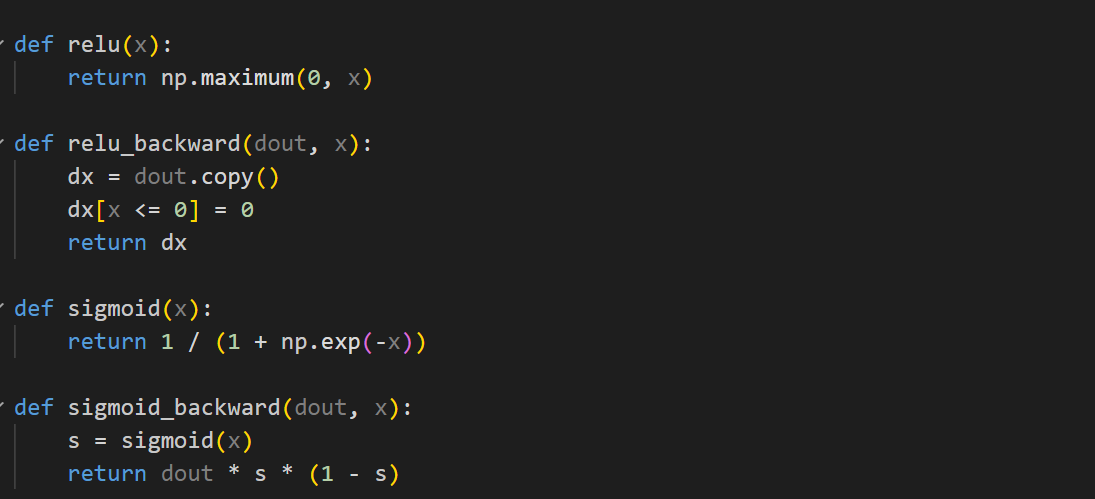
Działanie:

1. Gradient z wyjścia (dout) zostaje odpowiednio przycięty do oryginalnego rozmiaru (jeśli wcześniej został dopadnięty),
2. Następnie wykonywana jest standardowa backpropagacja konwolucji przy użyciu conv\_backward\_im2col,
3. Uzyskany gradient względem wejścia do warstwy dekonwolucji jest „ściągany” tylko z tych miejsc, które zawierały wartości w danych rozrzedzonych,
4. Gradienty względem wag są odwracane (flip) w celu uzyskania poprawnych wartości dla dekodera.

# Funkcje aktywacji i metryki

W projekcie zastosowano dwie podstawowe funkcje aktywacji oraz prostą metrykę do oceny jakości rekonstrukcji:

* **ReLU (Rectified Linear Unit)**  
  Funkcja aktywacji ReLU została użyta w części enkodera. Zwraca 0 dla wartości ujemnych oraz wartość bez zmian dla wartości dodatnich. Dzięki temu wprowadza nieliniowość, a jednocześnie jest bardzo wydajna obliczeniowo. Funkcja relu(x) realizuje aktywację, natomiast relu\_backward(dout, x) oblicza pochodną tej funkcji względem wejścia i przekazuje gradient dalej.
* **Sigmoid**  
  Funkcja aktywacji sigmoid została zastosowana w dekoderze, na końcu sieci. Jej zadaniem jest przeskalowanie wartości wyjściowych do zakresu [0, 1], co odpowiada normalizacji obrazu wejściowego. Funkcja sigmoid(x) oblicza aktywację, natomiast sigmoid\_backward(dout, x) realizuje obliczenie gradientu podczas backpropagacji.
* **binary\_accuracy**  
  Metryka binarnej dokładności porównuje oryginalny i zrekonstruowany obraz po binarnej progowej konwersji pikseli (zwykle przy progu 0.5). Zwraca wartość z przedziału [0, 1], która informuje, jaki procent pikseli został poprawnie odtworzony.



# Architektura autoenkodera

Autoenkoder składa się z dwóch głównych części:

* **Enkoder**
  + warstwa konwolucyjna z wieloma filtrami (np. 100 filtrów 3×3),
  + aktywacja ReLU,
  + uśrednianie wszystkich kanałów wzdłuż wymiaru filtrów (redukcja wymiarowości). Efektem enkodera jest jedna mapa cech (feature map) o tych samych wymiarach przestrzennych jak wejście.
* **Dekoder**
  + pojedynczy filtr transponowanej konwolucji uzyskany przez uśrednienie   
    wag enkodera,
  + aktywacja Sigmoid,
  + rekonstrukcja obrazu do pierwotnego wymiaru (64×64).

Parametry modelu:

* w\_enc – macierz wag enkodera o wymiarach (num\_filters, 3, 3),
* b\_enc – biasy enkodera,
* w\_dec – wagi dekodera uzyskane jako średnia wag enkodera,
* b\_dec – średnia biasów dekodera,
* stride – krok konwolucji (zwykle 1),
* pad – padding, umożliwiający zachowanie rozmiaru wejścia.

# Procedura treningowa

Uczenie autoenkodera odbywało się metodą pełnego batcha (na całym zbiorze jednocześnie), z wykorzystaniem ręcznie zaimplementowanej propagacji wstecznej.

**Funkcja straty**  
Zastosowano błąd średniokwadratowy (MSE), który mierzy średnią różnicę pomiędzy obrazem wejściowym a zrekonstruowanym.  
Funkcja mse\_loss(y\_true, y\_pred) zwraca zarówno wartość straty, jak i gradient po tej funkcji, niezbędny do aktualizacji parametrów.

**Pętla treningowa**  
Każda epoka uczenia składa się z:

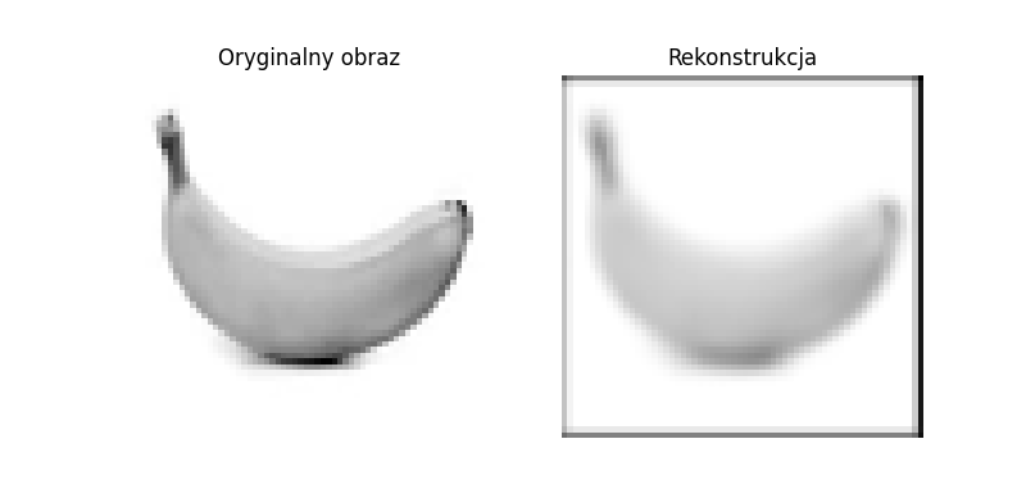
1. Obliczenia forward pass (przejście przez sieć),
2. Wyznaczenia wartości funkcji straty oraz metryki dokładności,
3. Obliczenia gradientów przy pomocy autoencoder\_backward,
4. Aktualizacji wag enkodera i dekodera zgodnie z regułą spadku gradientu.

**Ustawienia eksperymentalne**

* liczba filtrów: num\_filters = 100,
* liczba epok: epochs = 260,
* współczynnik uczenia: learning\_rate = 0.000005,
* inicjalizacja wag: losowa z rozkładu normalnego (np.random.randn(...) \* 0.1),
* ziarno generatora liczb losowych: np.random.seed(42).

# Wyniki i wizualizacja

Dla wybranych przykładów zaprezentowano wizualne porównanie oryginalnego obrazu z jego rekonstrukcją wygenerowaną przez autoenkoder po zakończeniu uczenia. Przykładowe wyniki wskazują, że sieć nauczyła się odtwarzać podstawowe struktury obrazu (kontury, kształty), mimo znacznej redukcji wymiarowości w warstwie ukrytej.



Rysunek Przykładowa rekonstrukcja

# Wyniki końcowe

* Projekt został zrealizowany zgodnie z założonym celem – zaimplementowano autoenkoder konwolucyjny od podstaw, bez użycia gotowych bibliotek do uczenia głębokiego. Zastosowane podejście umożliwiło pełną kontrolę nad procesem przetwarzania obrazu i uczenia modelu.
* Techniczna implementacja kluczowych komponentów (im2col, konwolucje, funkcje aktywacji, propagacja wsteczna) pozwoliła na lepsze zrozumienie działania sieci neuronowych w praktyce. Zastosowanie NumPy zapewniło przejrzystość i czytelność obliczeń.
* Wyniki testów potwierdziły, że nawet prosty autoenkoder jest w stanie nauczyć się odwzorowywać struktury zawarte w obrazie. Model osiągał wysoką dokładność binarną, a wizualnie generował rekonstrukcje zgodne z oryginałem.
* Możliwe zastosowania takiej architektury to m.in. kompresja danych obrazowych, odszumianie, wykrywanie anomalii czy przygotowanie warstw wstępnych do bardziej złożonych sieci (np. klasyfikatorów).
* Stopień realizacji celu należy ocenić jako pełny – zaimplementowano i uruchomiono wszystkie elementy, przeprowadzono eksperymenty, zebrano wyniki i sformułowano końcowe obserwacje.