
AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ**

# PRACA DYPLOMOWA INŻYNIERSKA

*Algorytm tworzenia profilu pisma odręcznego oparty o metody sztucznej inteligencji*

Autor: *Bartłomiej Szozda*
Kierunek studiów: Informatyka Stosowana
Opiekun pracy: *dr. inż. Szymon Łukasik*

Kraków, 2020

## Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu anty plagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....

(czytelny podpis)

Ocena merytoryczna Opiekuna

Ocena merytoryczna Recenzenta

Za inspirację, wyrozumiałość oraz pomoc przy realizacji tej pracy pragnę złożyć serdecznie podziękowania mojemu promotorowi dr. inż. Szymonowi Łukasikowi.

# Spis treści

# Wstęp

## Wstęp właściwy

Umiejętność ręcznego pisania jest dla nas ludzi bardzo ważna. Nawet w dobie elektronicznego zapisu danych często wolimy tworzyć i czytać odręczne notatki niż korzystać z klawiatury i pisma drukowanego. Ręczne pismo pozostaje najważniejszą metodą autentykacji – większość umów podpisywanych jest ręcznie, a szczególnie te najważniejsze, jak ogromne kontrakty czy porozumienia międzynarodowe. Uczymy się pisać od bardzo młodego wieku i potrzebujemy wiele praktyki, aby nasz mózg oswoił się z tą sztuką. Podobnym, lecz w zasadzie oddzielnym problemem jest naśladowanie cudzego pisma, bowiem wykształcenie własnego charakteru nie pozwala na odwzorowanie innych.

W pracy tej zmierzono się z problemem tworzenia odręcznego pisma za pomocą programu komputerowego. Wykorzystana będzie w tym celu sieć neuronowa tworząca obrazy zawierające znaki. Architekturą modelu będzie Generatywna Sieć Przeciwstawna (ang. generative adversarial networks, GAN) [1].

Generowanie odręcznego pisma może być użyte w:

- Generowaniu notatek wyglądających jak odręczne,
- Wypełnianiu brakujących fragmentów tekstu w dziełach literackich lub źródłach historycznych,
- Poszerzaniu zbiorów danych zawierających takie pismo,
- Fałszowaniu podpisów.

Ostatnie wymienione zastosowanie jest niewątpliwie negatywnym aspektem. Jednak prace naukowe mają tutaj pozytywny wydźwięk. Po pierwsze publicznie pokazują jakie możliwości mogą mieć oszuści. Po drugie mogą służyć jako źródło fałszywych danych dla programu rozpoznającego czy pismo jest autentyczne.

Proces uczenia użytej sieci potrzebuje przeanalizować ogromne ilości przykładów. Aby zminimalizować ilość możliwych kombinacji ograniczono się do generowania par liter, co było potrzebne ze względu na skończoną ilość mocy obliczeniowej oraz czasu. Wciąż jednak pozostaje możliwe wygenerowanie ciągłego pisma, gdzie łączenie pomiędzy literami będzie realistyczne. Zdecydowano się położyć nacisk na realizm tworzonego pisma, zamiast np. sklejać ze sobą pojedyncze wygenerowane litery.

Tworzenie par zamiast pojedynczych liter niesie za sobą wiele problemów. Największym z nich jest brak możliwości zdobycia gotowego zbioru danych. Ręczne wygenerowanie nie wchodzi w grę, gdyż sieć neuronowa potrzebuje około 6 tysięcy przykładów każdej pary, a możliwych par w alfabecie łacińskim jest 676. Zmierzono się z tym problemem w rozdziale 3.

## Problem w nazewnictwie

Sieć na której oparto rozwiązanie problemu nazywać będziemy siecią główną lub siecią GAN.

W celu stworzenia zbioru danych skorzystano z generatora tekstu opierającego się także na sieci neuronowej lecz innej architektury. Innym generatorem jest część sieci głównej. Warto zaznaczyć, że często używane pojęcia ‘generator’ oraz ‘generować’ będą miały różne znaczenia. Słowo ‘generator’ dotyczy albo generatora za pomocą którego tworzone były dane wejściowe dla głównej sieci, albo generatora należącego do głównej sieci. Natomiast ‘generować’ odnosi się czasem do danych wejściowych tworzonych przez generator, albo do danych wyjściowych otrzymanych poprzez użycie generatora z głównej sieci.

# Cel pracy

Celem pracy jest stworzenie za pomocą komputera par liter (z alfabetu łacińskiego) wyglądających jak pisane ludzką ręką. Wykonane to będzie przy użyciu sieci neuronowej o architekturze GAN. Realizm podczas tego zadania był na tyle ważny, ze zrezygnowano z prostszego problemu generowania pojedynczych liter i ewentualnego tworzenia tekstu poprzez zbliżanie ich do siebie, na rzecz tworzenia par liter z naturalnie wyglądającym łączeniem.

Praca ta mogłaby w przyszłości pomóc w generowaniu całego tekstu, np. poprzez:

- algorytm łączący pary ‘ad’, ‘da’ w wyraz ‘ada’. Algorytm taki powinien mieć ułatwione zadanie w porównaniu do łączenia typu ‘a’, ‘b’ w wyraz ‘ab’, gdyż posiada literę która może być pomocna w złączeniu dwóch par liter.
- rozszerzenie funkcjonalności programu do całych wyrazów. Zadania tego nie podjęto się od razu ze względu na jego trudność oraz prawdopodobną potrzebę bardzo długiego czasu obliczeniowego.

Praca jest wyjściem poza generowanie pojedynczych znaków i może dać doświadczenia przydatne w dalszym rozszerzaniu funkcjonalności (całe wyrazy) lub być jej częścią (łączenie par w wyrazy).

# Motywacja

## Potencjał uczenia maszynowego i sieci GAN:

Uczenie maszynowe (ang. machine learning, ML) jest niezwykle prężnie rozwijającą się dziedziną nauki. Znajduje zastosowania w bardzo wielu aspektach i potrafi rozwiązać coraz więcej problemów. Dla przykładu wymienić można tutaj takie dziedziny jak:

- Opieka zdrowotna (wykrywanie chorób, tworzenie leków)
- Wykrywanie oszustw
- Przewidywanie zmian na rynkach finansowych
- Zrozumienie klienta (rekomendacje, przewidywanie potrzeb)
- Przetwarzanie języka (mówionego, pisanego)
- Inteligentne urządzenia (np. samochody)

Powyższa lista zastosowań jest w rzeczywistości ogromna. Uczenie maszynowe może przydać się wszędzie tam, gdzie potrzeba przetworzyć jakieś dane, a świat generuje ich coraz więcej. Nie jest tajemnicą, że największe korporacje takie jak Google, Facebook, Microsoft czy Apple przykładają szczególną uwagę do posiadania oraz przetwarzania informacji. Niemal pewne jest, że znaczenie danych tak samo jak technik ich przetwarzania będzie w przyszłości tylko większe.

Wykorzystana w tej pracy sieć neuronowa (GAN) należy do modeli generatywnych (ang. generative models, GM) które z kolei należą do dziedziny uczenia bez nadzoru (ang. unsupervised learning). W uczeniu takim posiadamy dane wejściowe, ale nie mamy konkretnego przykładu co z danymi ma się stać. Celem GM jest ‘poznanie’ danych wejściowych i na ich podstawie tworzenie nowych odpowiedniego typu. Można powiedzieć, że modelem przeciwstawnym do GM jest model dyskryminacyjny (ang. discriminative models, DM), zajmujący się klasyfikowaniem danych. Dla ludzi stworzenie czegoś jest trudniejsze niż rozpoznanie i analogiczną zasadę możemy zaobserwować dla sieci neuronowych - zastosowania modeli GM przynosiły małe rezultaty w porównaniu do DM. Duży postęp w sieciach GM przyniósł opisywany tutaj model GAN.

O potencjale sieci GAN świadczyć może natomiast ogromny wzrost oraz utrzymanie się zainteresowania tym rodzajem sieci (na podstawie Google Trends) zaprezentowany na rysunku 1.


**Rysunek 1 Wzrost ilości wyszukiwań hasła "Generative Adversarial Networks" wg. Google Trends.**

**Wykres pokazuje stosunek, nie ilość. Przykładowo 100 (maksymalna liczba wyszukiwań), oznacza 4 razy więcej wyszukiwań niż 25.**

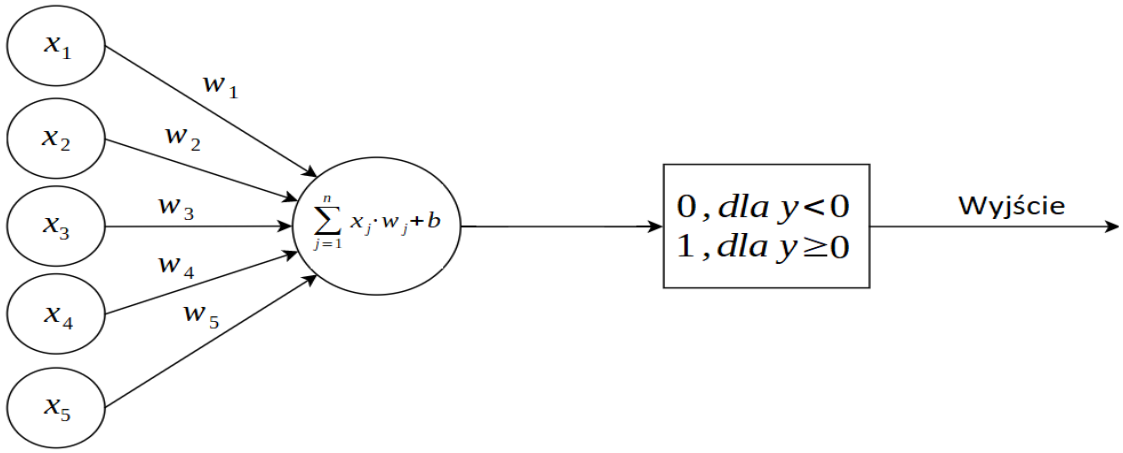
## Brak publikacji o GAN wykraczających poza pojedyncze znaki:

Projekt do tej pracy zaczął powstawać już w drugiej połowie 2018 roku. Pomimo wysokiego zainteresowania sieciami GAN nie odnaleziono wtedy prac dotyczących generowania ręcznego pisma za pomocą sieci GAN, nie licząc tworzenia pojedynczych znaków. W pracy tej zdecydowano się pójść o krok dalej, tworząc rozwiązanie pośrednie pomiędzy pojedynczymi znakami, a całymi wyrazami tj. generowanie par liter. Prawdopodobnie pierwsza praca opisująca tworzenie wyrazów za pomocą sieci GAN opublikowana została dopiero w marcu 2019 roku [2].

# Rozdział 1 Wprowadzenie teoretyczne

## 1.1 Perceptron

Perceptron to klasyfikator binarny, ponieważ potrafi sklasyfikować dane wejściowe przypisując je do dwóch klas (0 lub 1). Wymyślony został w 1958 roku przez Franka Rosenblatta [3].




**Rysunek 2 Perceptron, użyto [44].**

Na wejściu dane ($x_1, x_2, \ldots x_5$) które chcemy sklasyfikować binarnie zwracając wartość wyjściową (0 lub 1). Czasem do wartości sumy dodawane jest obciążenie (ang. bias) oznaczone jako b. Wartość ta zależy od równania (funkcja Heaviside'a):

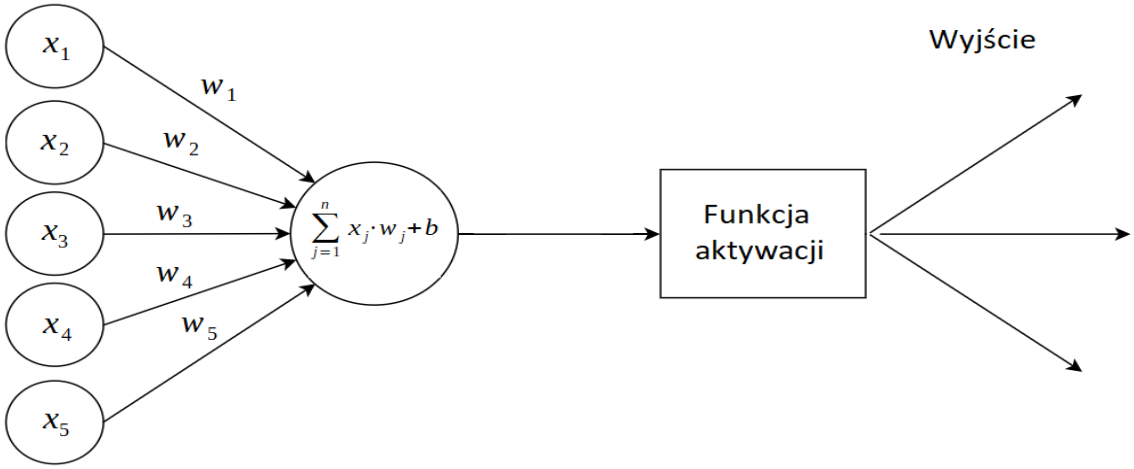
$$\begin{matrix} 0, dla\ y<0 \\ 1, dla\ y\geq 0 \end{matrix}$$

gdzie y oznacza sumę

$$\sum_{j=1}^{n} x_j \cdot w_j + b$$

## 1.2 Neuron

Bardzo podobny do perceptronu.




**Rysunek 3 Neuron, użyto [44].**

Różnice względem perceptronu:

- Zamiast funkcji Heaviside'a możemy zastosować funkcję aktywacji (opisane w rozdziale 1.3).
- Może mieć wiele wyjść do kolejnych neuronów.

## *1.3 Funkcje aktywacji*

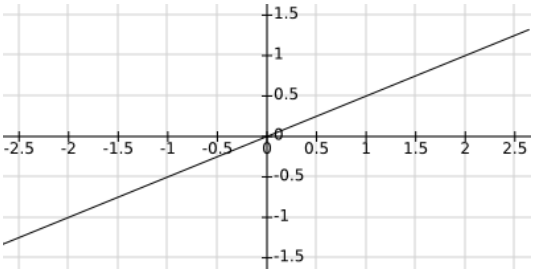
Służy do obliczenia wartości która będzie zwrócona przez neuron.

Wymienione zostanie tu kilka przykładowych funkcji aktywacji:

### Liniowa:

Przenosi sygnał (może także wzmacniać lub wygaszać sygnał). Z reguły b=0 [4].

$$y = a \cdot x + b$$

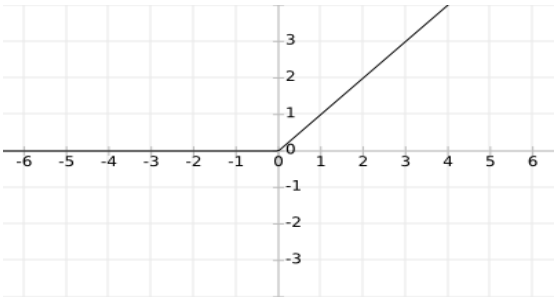



**Rysunek 4 Liniowa funkcja aktywacji, wygenerowana za pomocą [45]**

### ReLU:

$$0, dla\ x < 0$$
$$x, dla\ x \geq 0$$

Najczęściej używany w głębokich sieciach neuronowych [5].

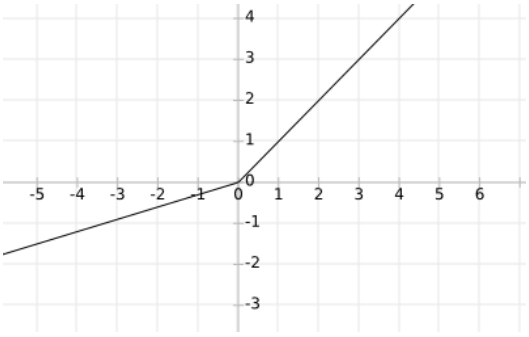



**Rysunek 5 Funkcja aktywacji ReLU, użyto [45]**

## Leaky ReLU:

Podobna do ReLU lecz posiada współczynnik nieszczelności **a** (stosuje się małe wartości **a**)

$$ax, dla\ x<0$$
$$x\ \ , dla\ x\geq 0$$

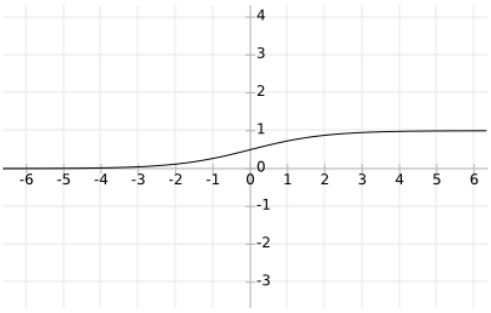



**Rysunek 6 Funkcja aktywacji leaky ReLU, użyto [45]**

## Sigmoidalna:

Zwraca wartości z przedziału [0,1] przez co może być używana do zwrócenia wyniku będącego prawdopodobieństwem.

$$\frac{1}{1+e^{-x}}$$

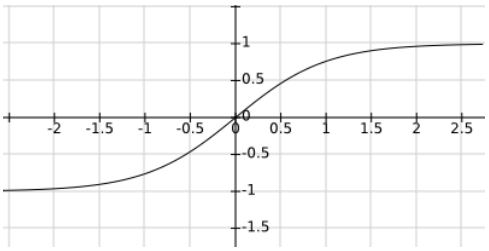



**Rysunek 7 Sigmoidalna funkcja aktywacji, użyto [45]**

## Tangens hiperboliczny:

Zwraca wartości z przedziału [-1,1].

$$\frac{1-e^{-x}}{1+e^{-x}}$$




**Rysunek 8 Funkcja aktywacji tangens hiperboliczny, użyto [45]**

## *1.4 Uczenie maszynowe*

Uczenie maszynowe (ang. machine learning, ML) jest częścią sztucznej inteligencji (ang. artifical inteligence). Dotyka kilku dziedzin takich jak informatyka, statystyka, matematyka. Potrafi rozwiązywać problemy które nie są możliwe do rozwiązania tradycyjnymi metodami. Algorytmy tego typu udoskonalają się wraz z przebiegiem nauki, a do ich działania zazwyczaj niezbędny jest duży zbiór danych. Wyróżniamy kilka typów uczenia maszynowego:

Uczenie z nadzorem (ang. supervised learning):

Używane przede wszystkim do nauki klasyfikacji. Dane wejściowe oznaczone są etykietami (oznaczającymi klasę). Algorytm przetwarza dane i tworzy metodę zwracającą etykietę. Zazwyczaj nauka polega na przeanalizowaniu jak dobry wynik zwróciła metoda (poprzez porównanie zwróconej wartości, do etykiety przypisanej wcześniej) i zmiany struktury algorytmu za pomocą specjalnych technik (rozdział 1.6). W ten sposób algorytm uczy się wyciągać cechy szczególne na podstawie których przypisuje etykiety. Celem jest pozyskanie zdolności do klasyfikacji nowych danych, nieposiadających etykiet.

Nazywa się uczeniem z nadzorem, gdyż wymaga ingerencji w proces nauki poprzez przypisane wcześniej etykiety, co musi być zrobione za pomocą innej techniki, np. poprzez człowieka.

Uczenie bez nadzoru (ang. unsupervised learning):

Inaczej niż podczas uczenia z nadzorem - nie dostarczamy etykiet. Zadaniem algorytmu jest samodzielne wyszukanie cech charakterystycznych oraz podobieństw i dokonanie tzw. klasteryzacji, czyli podzielenia zbioru danych na kategorie. Nazywane jest uczeniem bez nadzoru, gdyż nie wymaga zewnętrznej ingerencji w czasie nauki.

Uczenie pół-nadzorowane (ang. semi-supervised learning):

Jest mieszanką obydwu wymienionych wcześniej sposobów uczenia. Jedna część danych posiada etykiety, druga nie. Algorytmy tego typu uczą się jednocześnie klasyfikacji i klasteryzacji.

Uczenie ze wzmocnieniem (ang. reinforcement learning):

W przeciwieństwie do poprzednich metod algorytm nie uczy się na podstawie przygotowanych wcześniej danych. Informacje potrzebne do procesu uczenia pozyskiwane są na bieżąco. Algorytm uczy się interakcji z jakimś środowiskiem i podejmowania na bieżąco najlepszych akcji. Podjęta akcja oceniana jest na podstawie zdefiniowanych przez twórcę zasad. Ocena ta pełni funkcję nagrody lub kary i nazywana jest wzmocnieniem. Na jego podstawie algorytm uczy się podejmować odpowiednie decyzje, gdyż stara się maksymalizować zysk.

## *1.5 Sieć neuronowa*

### 1.5.1 Opis ogólny

Sieć neuronowa (ang. neural network, NN) to pewien model w jakimś stopniu wzorowany na ludzkim mózgu, służący do rozwiazywania problemów. Przynależy do uczenia maszynowego i boryka się z zadaniami dla których tradycyjne algorytmy nie przynoszą rozwiązań. Jest siecią połączonych ze sobą neuronów (rozdział 1.2). Posiada warstwy do których należą poszczególne neurony i które możemy podzielić na trzy rodzaje

1. Warstwa wejściowa.

   Przyjmuje dane pochodzące np. ze zbioru danych. Przetwarza je i przekazuje je do następnej warstwy (ukrytej lub wyjściowej)

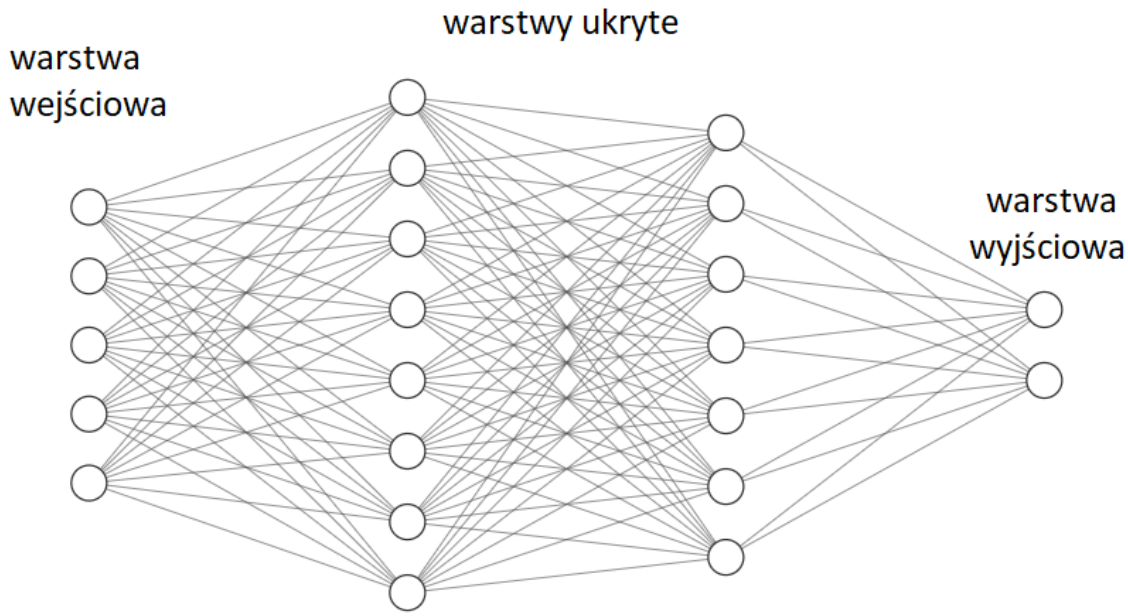
2. Warstwa ukryta.

   Nie musi być obecna, ale może być też wielokrotna. Przyjmuje wartości przekazane przez poprzednią warstwę, przetwarza je i przekazuje do następnej warstwy (ukrytej lub wyjściowej)

3. Warstwa wyjściowa.

   Przyjmuje wartości z poprzedniej warstwy, przetwarza i przekazuje na wyjście. Dane takie traktowane są jako wynik działania modelu.

Przykładową sieć neuronową zobrazowano na poniższym rysunku:




**Rysunek 9 Sieć neuronowa posiadająca 4 warstwy, użyto [46]**

Sieć ta posiada 4 warstwy. Wejściowa (5 neuronów), dwie warstwy ukryte zawierające odpowiednio 8 oraz 7 neuronów oraz warstwę wyjściową (2 neurony).

Warto nadmienić, że jest mnóstwo rodzajów sieci neuronowych. Dla przykładu wymieniono kilka podziałów:

- Z przepływem informacji w jedną stronę, bez sprzężeń zwrotnych (ang. feedforwad neural networks, FNN)
- Ze sprzężeniami zwrotnymi (ang. recurrent neural networks, FNN)
- Posiadające więcej warstw ukrytych, nazywane głębokimi(ang. deep neural networks, DNN).
- Konwolucyjne, z powodzeniem stosowane w przetwarzaniu większych obrazów (ang. convolutional neural networks, CNN)

Poszczególne architektury mogą należeć do kilku typów. np. głęboka, konwolucyjna sieć neuronowa z przepływem informacji w jedną stronę (DNN, CNN oraz FNN). Natomiast niektóre rodzaje są sobie przeciwne i nie mogą występować razem (np. FNN oraz RNN).

### 1.5.2 Głębokie sieci neuronowe

W pracy tej użyteczna będzie definicja głębokich sieci neuronowych z przepływem informacji w jedną stronę (DNN oraz FNN). Również opisane tu zagadnienia przydatne będą w dalszej części pracy.

Na rysunku 9 przedstawiono niewielką głęboką sieć neuronową z 2 warstwami ukrytymi zawierającymi po kilka neuronów. Warstw tych może być jednak więcej, a każda z nich zawierać może zdecydowanie wyższą liczbę neuronów. Biorąc za przykład problem klasyfikacji obrazu, pierwsze warstwy ukryte potrafią wydobywać najprostsze cechy z sygnałów wejściowych. Kolejne bazując na odpowiedzi poprzedników reprezentują coraz to ogólniejsze cechy, aż do warstwy wyjściowej z której odczytujemy wynik klasyfikacji.

Sieci tego typu mogą być bardzo skuteczne, ich odpowiednie użycie zrewolucjonizowało tematykę analizy obrazów. Mają też swoje wady – często do nauki potrzebują przeanalizować ogromne ilości danych, do czego wymagane są spore moce obliczeniowe, a sam czas uczenia jest długi.

Struktura sieci DNN będąca często bardziej skomplikowaną niż struktura sieci płaskich powoduje, że są one bardziej narażone na problem przeuczenia (ang. overfitting). Dlatego często stosowane są techniki regularyzacji, np.technika tzw. odpadania (ang. dropout). Obydwa te pojęcia wyjaśnione będą w kolejnych rozdziałach (1.6.3 Przeuczenie oraz 1.6.4 Technika regularyzacji - odpadanie).

## *1.6 Uczenie sieci neuronowej*

### 1.6.1 Przygotowanie danych

Opisywany proces skupiał się będzie na uczeniu nadzorowanym.

W pierwszej kolejności należy przygotować zbiór danych wraz z etykietami. Następnie dzielimy go na trzy części zawierające np. 60%, 20%, 20% ogółu elementów. Pierwszy z nich będzie zbiorem treningowym, drugi walidacyjnym, a trzeci testowym. Zbiór treningowy używany jest do nauki sieci, na jego podstawie ustawiane są odpowiednie parametry. Zbiór walidacyjny stosowany jest w czasie treningu, lecz nie jest używany do nauki sieci. Wykorzystuje się go głównie w celu sprawdzania czy nie występuje przeuczenie (1.6.3 Przeuczenie). Natomiast zbiór testowy nie jest używany w czasie trenowania i służy do ostatecznej weryfikacji skuteczności programu.

### 1.6.2 Proces uczenia

Po przygotowaniu zbioru danych następuje proces uczenia.

Pojedyncza iteracja tego procesu:

- Dane przekazywane są do warstwy wejściowej.
- Przetwarzane są one przez kolejne warstwy co nazywane jest propagacją w przód (ang. forward propagation).
- Warstwa wyjściowa zwraca wynik będący klasyfikacją.
- Rezultat ten porównywany jest z oczekiwanym, a błąd dopasowania opisujący jak bardzo sieć się myliła obliczany jest za pomocą funkcji straty (ang. loss function), im wynik bliższy zera tym różnica była mniejsza.
- Następuje proces zwany propagacją wsteczną (backpropagation). Ma on odwrotny kierunek niż w przypadku propagacji w przód. W wielkim skrócie zadaniem tego procesu jest przypisanie poszczególnym neuronom jak bardzo wpłynęły na otrzymaną stratę.
- Zmiana wartości wag poszczególnych neuronów. Korzysta się w tym celu z algorytmu optymalizacyjnego np. z metody gradientu prostego (ang. gradient descent) [6] mającego na celu zminimalizowanie straty, czyli wykonanie kroku uczącego.

W czasie trwania nauki sieci wykonywanych jest mnóstwo iteracji procesu uczącego.

### 1.6.3 Przeuczenie

W procesie uczenia zazwyczaj chodzi o to, aby model używając danych treningowych, nauczył się ogólnych cech tego typu zbioru i mógł być zastosowany dla innych danych. Problem przeuczenia (nadmiernego dopasowania, przetrenowania) w sieciach neuronowych to zbyt duże dopasowanie parametrów do rozpoznawania danych treningowych i brak zdolności generalizacji. Obserwujemy to, kiedy w czasie nauki model doskonale radzi sobie z postawionym zadaniem, natomiast jest mało użyteczny w przypadku danych innych niż treningowe.

Do przeuczenia może dojść, kiedy czas trwania nauki był zbyt długi, lub kiedy model sieci jest zbyt skomplikowany w stosunku do ilości danych. Przeuczony model zaczyna być zbyt dokładny wyodrębniając cechy pasujące tylko do danych, za pomocą których był uczony.

Odwrotnym problemem jest niedouczenie, występujące, gdy zbiór danych jest zbyt mały lub użyty model zbyt prosty.

### 1.6.4 Technika regularyzacji - odpadanie

Jedną z technik zapobiegania przeuczeniu jest tzw. technika odpadania (ang. dropout [7]). Zastosowana dla danej warstwy powoduje, że niektóre z jej neuronów są wyłączane. Każdy neuron wyłączany jest lub nie, zgodnie z zadanym prawdopodobieństwem. Pozbywanie się jakiejś części neuronów, to tak naprawdę zmniejszenie informacji przekazywanych z jednej warstwy do drugiej. Zmusza to sieć do nabycia umiejętności rozpoznawania używając zmniejszonej ilości danych oraz nieprzywiązywania się do

wszystkich wyodrębnianych cech zbioru treningowego. Technika odpadania jest jedną z najczęściej i najlepiej stosowanych technik regularyzacji.

### 1.6.5 Różne okresy wykonywania optymalizacji

Nie koniecznie optymalizacja wag (propagacja wsteczna oraz algorytm optymalizacyjny) wykonywana jest co każdy egzemplarz danych wejściowych. Stosuje się kilka typów algorytmu gradientu prostego [6]:

- Stochastyczny gradient prosty (ang. stochastic gradient descent).

Optymalizacja wag wykonywana co każdy egzemplarz, tak jak w przykładzie. Duża fluktuacja wartości funkcji straty (także wokół minimum), lecz szybko zmierza do minimum. Ze względu na pojedynczy egzemplarz obliczenia nie podlegają procesowi wektoryzacji [8]

- Wsadowy gradient prosty (ang. batch gradient descent).

Optymalizacja wag wykonywana raz na okres przetworzenia wszystkich danych. Powolniejszy, lecz nie ma fluktuacji. Użycie jest problematyczne, kiedy zbiór danych jest ogromny.

- Gradient prosty dla mini-grup (ang. mini-batch gradient descent).

Optymalizacja wag wykonywana raz na zadaną ilość egzemplarzy zwanych mini-grupą. Można w ten sposób zniwelować wady powyższych sposobów - zmniejszyć fluktuację i pozwolić na wykorzystywanie procesu wektoryzacji.

### 1.6.6 Binarna entropia krzyżowa

W pracy tej jako funkcji straty użyto binarnej entropii krzyżowej (ang. binary cross-entropy) [9].

Klasyfikacja binarna dotyka problemu w którym:

- Każdy egzemplarz należy do jednej z dwóch klas (w naszym problemie będzie to para liter lub nie)
- Każdy egzemplarz jest niezależny od innego.
- Wszystkie egzemplarze należą do tego samego rozkładu (para liter lub nie)

Binarna entropia krzyżowa może być opisana wzorem:

$$l = -\left[y \cdot \log(x) + (1-y) \cdot \log(1-x)\right],$$

gdzie y oznacza wartość oczekiwaną, a x wartość podaną.

## *1.7 Modele generatywne*

Opisywana wcześniej metoda uczenia sieci skupiała się na modelu dyskryminacyjnym (ang. discriminative model, DM). Model taki zdolny jest do określenia prawdopodobieństwa, że podane do niego dane (cechy) należą do obiektu jakiejś klasy. Może w ten sposób np. odpowiedzieć na pytanie, czy podany obrazek jest parą liter.

Model generatywny (ang. generative models, GM) chce dokonać czegoś innego - określić rozkład prawdopodobieństwa wystąpienia poszczególnych cech dla obiektu należącego do jakiejś klasy[10]. Rozkład taki możemy wykorzystać w celu generowania nowych danych należących do danej klasy.

Modele generatywne możemy podzielić na dwie główne kategorie, w zależności od tego jak definiowany jest wymieniony wyżej rozkład prawdopodobieństwa: modele jawnie definiujące (ang. explicit density models, EDM) oraz modele niejawnie definiujące (ang. implicit density models, IDM). Podkategorią modeli IDM są modele bezpośrednie (ang. DM - direct models, DM). Oprócz niejawnego definiowania prawdopodobieństwa potrafią one za jego pomocą generować nowe dane. Z kolei podkategorią modeli DM jest klasa algorytmów zaproponowanych niedawno, bo w 2014 roku, tj. generatywne sieci przeciwstawne (ang. generative adversarial networks, GAN) [1]

## *1.8 Generatywne sieci przeciwstawne*

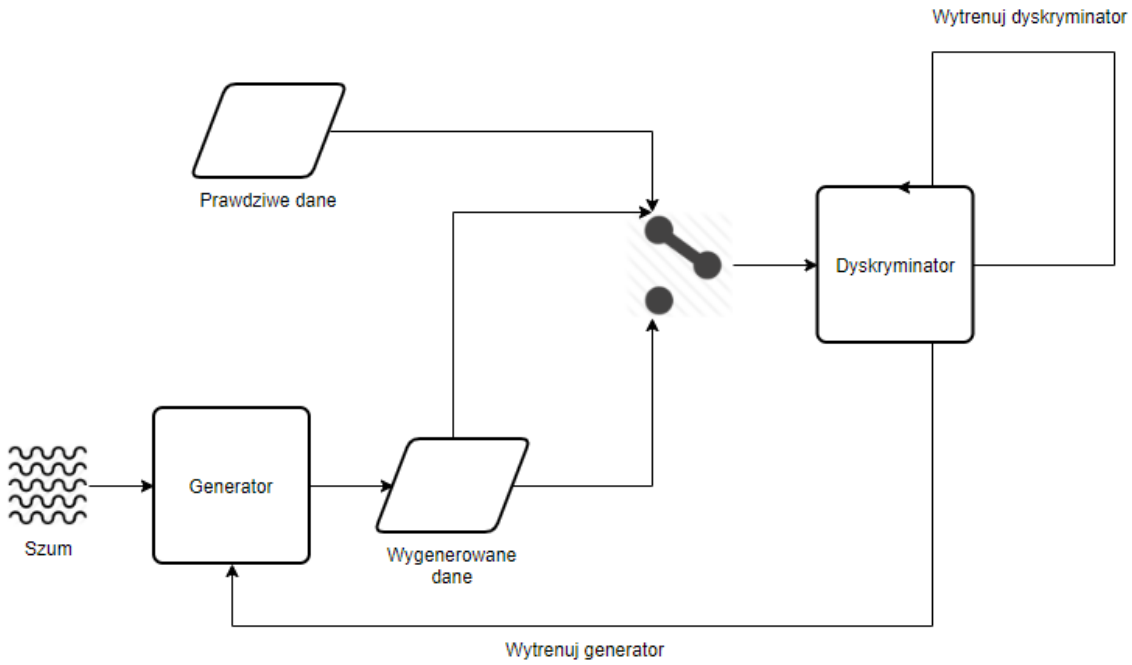
### 1.8.1 Sposób działania

Jak wspomniano w poprzednim podrozdziale, GAN należy do modeli generatywnych niejawnie definiujących prawdopodobieństwo i potrafiących generować dane.

GAN składa się z dwóch sieci, jednej nazywanej dyskryminatorem oraz drugiej nazywanej generatorem. Pierwsza z nich zwraca prawdopodobieństwo z przedziału [0,1] opisujące, czy podany do niej obiekt jest prawdziwy, Druga wytwarza nowe obiekty.

Sieci te współzawodniczą ze sobą, tj. generator próbuje wytworzyć dane tak, aby dyskryminator uznał je za prawdziwe. Dyskryminator z kolei doskonali się analizując prawdziwe oraz wygenerowane dane. Wraz z przebiegiem nauki generator wytwarza coraz to lepsze dane, aż stają się one nierozróżnialne z prawdziwymi. Opisany tutaj proces zobrazowany jest na rysunku 10.

Dobrym oraz popularnym przykładem opisującym proces nauki sieci jest porównanie generatora do fałszerza dzieł sztuki, natomiast dyskryminatora do badacza sztuki. Na początku obydwoje są niedoświadczeni. Badacz doskonali się określając prawdopodobieństwo czy dane dzieła sztuki są prawdziwe, czy fałszywe. W zależności od tego jak bardzo myli się w klasyfikacji, tak bardzo poprawia swoje metody. Fałszerz dąży do oszukania badacza, zatem doskonali swoje metody w zależności od stopnia z jakim badacz wykrył fałszerstwo.




**Rysunek 10 Schemat pracy sieci typu GAN, użyto [44]**

### 1.8.2 Gra minimax

Problem ten sprowadzić można do dwuosobowej gry minimax (wzorując się na twórcach architektury [1]).

$$\min_G \max_D V(D,G) = E_{x \sim pdata(x)}[\log D(x)] + E_{z \sim pz(z)}[\log(1 - D(G(z)))],$$

$$gdzie:$$

$$\min_G \max_D - minimalizacja\ największej\ straty$$

$$V(D,G) - funkcja\ zwrotu$$

$$E_{w \sim p(w)}[f(w)] - wartość\ oczekiwana\ funkcji\ f(w), gdzie\ w\ pochodzi\ z\ rozkładu\ p(w)$$

$$pdata(x) - rozkład\ gęstości\ danych\ z\ prawdziwego\ zbioru$$

$$pz(z) - rozkład\ gęstości\ szumu$$

$$D(x) - sieć\ dyskryminatora.\ Zwraca\ prawdopodobieństwo\ z\ przedziału\ [0:1]$$

$$G(x) - sieć\ generatora,\ zwraca\ nowe\ obiekty$$

Celem dyskryminatora będzie maksymalizowanie

$$\log D(x) + \log(1 - D(G(z)))$$

Celem generatora będzie z kolei minimalizowanie:

$$\log(1 - D(G(z)))$$

### 1.8.3 Szczegółowy proces działania

Proces uczenia został opisany przez twórców architektury GAN w następujący sposób:

**for** number of training iterations **do**

**for** $k$ steps **do**

- Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

**end for**

- Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Źródło [1]

# Rozdział 2 Wykorzystane technologie

W pracy tej starano się na bieżąco zwracać uwagę na użyte technologie. W rozdziale tym zostaną one tylko po krótce wymienione.

## Język Python wersje 3.6 oraz 2.7

Powszechnie dostępne poradniki, mnóstwo użytecznych bibliotek i narzędzi. Najpopularniejszy do zastosowań związanych z uczeniem maszynowym i sztuczną inteligencją.

## Najważniejsze użyte moduły (biblioteki) z języka Python:

- NumPy
- Matplotlib
- Pillow
- Multiprocessing

## PyTorch:

Biblioteka wykorzystywana do rozwiązań typu uczenie maszynowe i sztuczne sieci neuronowe. Podobne zastosowania co bardzo popularna biblioteka TensorFlow. Użyta do implementacji sieci głównej, wybrana na podstawie własnych odczuć jako bardziej przyjaznej dla programisty.

## Jupyter Notebook:

Interaktywny notatnik dla języka Python za pomocą którego można tworzyć rozwiązania, uruchamiać programy i przeglądać wyniki.

## TensorBoardX:

Użyty do obrazowania procesu uczenia generatora, poprzez wyświetlanie tworzonych obrazów w notatniku Jupyter Notebook.

## System operacyjny:

Windows 10 wraz z pakietem Windows Subsystem For Linux (skrót WSL). Większość procesów uruchamiana z poziomu WSL, czyli jako Linux-owe programy wykonywalne.

Google Cloud Platform:

Usługa obliczeń w chmurze (stosowana przede wszystkim do wygenerowania zbioru danych)

Pl-Grid, superkomputer Zeus:

niestety tylko próby użycia.

TensorFlow, OpenCV:

Użyte w oprogramowaniu pomocniczym, czyli np. sieciach służących jako generator zbioru danych oraz OCR (rozwiązania stworzone przez kogoś innego, dostosowane na własny użytek).

# Rozdział 3 Przygotowanie danych

## 3.1 *Ilość danych i ich format*

Sieć powinna generować pary liter stworzone za pomocą alfabetu łacińskiego i zawierające powtórzenia. Alfabet ten ma 26 znaków, a liczba par będzie odpowiadać liczbie kombinacji 2-elementowych z powtórzeniami na 26-elementowym zbiorze i wynosi $26 * 26 = 676$.

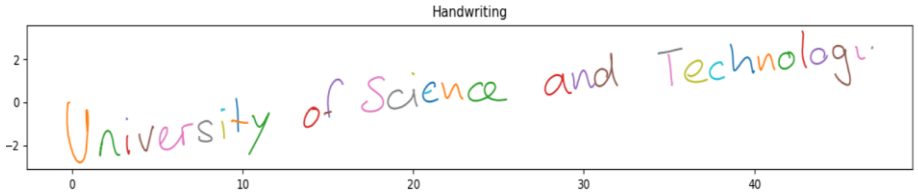
W celu oszacowania rozmiaru oraz liczby poszczególnych par przestudiowano artykuły dotyczące generowania cyfr [11] czy liter [12] . W pracach tych wykorzystano popularne zbiory odpowiednio dla cyfr: MNIST [13] oraz dla liter: eMNIST [14]. Zarówno MNIST jak i eMNIST zawierają około 6 tysięcy obrazków każdej litery lub znaku. Na tej podstawie przyjęto, że zestaw danych powinien zawierać 6 tysięcy egzemplarzy dla każdej generowanej pary, co łącznie daje przynajmniej 2 miliony obrazków.

Wybranie rozmiaru danych wejściowych jest ważnym problemem, gdyż jednocześnie określa rozmiar danych generowanych przez algorytm, czyli par znaków pisanych ludzkim stylem. Rozmiar ten nie może być zbyt mały (nieczytelne wyniki) ani zbyt duży (dłuższy proces uczenia). Ponownie przyjrzano się popularnym zbiorom MNIST i eMNIST, które to korzystają z obrazków o wymiarach 28x28 pikseli. Zdecydowano, że rozmiar ten będzie wystarczająco czytelny, a jednocześnie nie obciąży programu zbyt dużą ilością obliczeń.

## 3.2 *Próby pozyskania zbioru*

Zgodnie z tym co wcześniej obliczono do właściwej analizy rozważanego zagadnienia potrzeba przynajmniej 2 miliony obrazków zawierających pary liter, dlatego ręczne stworzenie zbioru jest wykluczone. W związku z tym dokonano prób znalezienia zbioru danych zawierającego pary liter pisane ludzkim charakterem pisma. Niestety poszukiwania zakończyły się fiaskiem. Możliwe było jedynie odnalezienie zestawów danych zawierających pojedyncze litery, całe wyrazy albo całe linijki.

Przeanalizowano możliwość wycinania par liter z odnalezionych zbiorów. Ludzkie pismo jest jednak bardzo różnorodne co nie pozwoliło na proste rozwiązanie problemu. Podjęto próby wyszukania odpowiedniego algorytmu, jednak w międzyczasie znaleziono pracę naukową [15], a następnie jej implementację [16], czyli generator tworzący pismo bardzo podobne do ludzkiego.




Rysunek 11: Przykładowe pismo stworzone przez generator

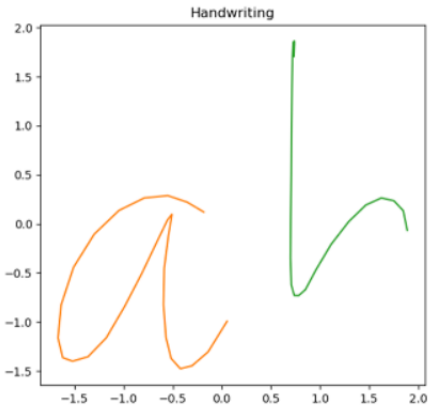
Poszczególne kolory odpowiadają pojedynczym pociągnięciom pióra. Można zauważyć, że niektóre pary znaków pisane są razem, dokładniej: ‘er’, ‘ci’, ‘ce’. Jest to pożądana cecha, gdyż nadaje realizmu tworzonemu zbiorowi. W założeniach pracy odrzucono pomysł sztucznego sklejania pojedynczych liter w pary aby zachować jak największą naturalność tworzonego pisma. Zbiór danych tworzony przez generator [16] powinien pozwolić spełnić to założenie.

## *3.3 Generowanie danych wejściowych*

### 3.3.1 Generowanie poszczególnego egzemplarza

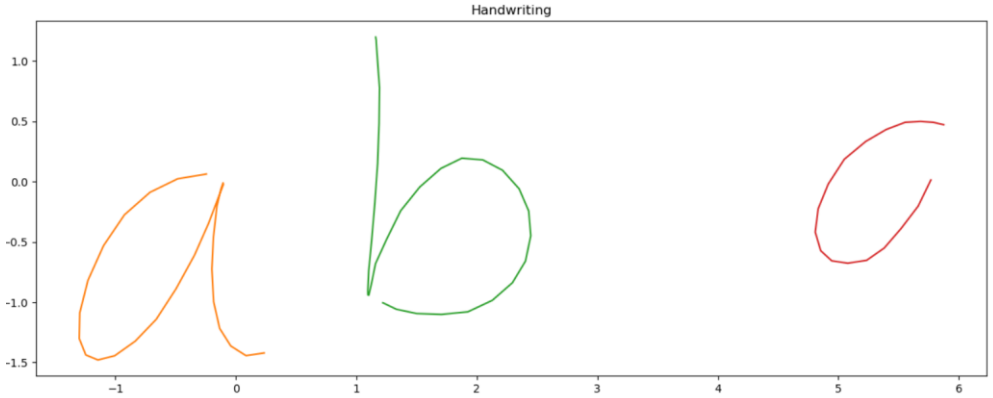
Ciąg znaków wygenerowany przez generator to zbiór danych odpowiadających pociągnięciom pióra (różne kolory na rysunku 11). Każde z tych pociągnięć wyrażone zostało jako zbiór dwuwymiarowych punktów.

Niestety generator [16] często ma problem z poprawnym zapisem ostatniej litery. Czasem zdaje się, że nie zawsze generuje ostatnie pociągnięcie piórem (niepoprawne ‘y’ na samym końcu napisu z rysunku 11), jednak w ogromnej części przykładów brakujący fragment nie powinien być dodatkowym pociągnięciem pióra (urwana w połowie litera ‘b’ z rysunku 12)




Rysunek 12: Wygenerowane „ab”

Nie udało się dotrzeć do przyczyny tego problemu. Zdecydowano się generować dłuższy napis, aby wyciąć z niego poprawnie stworzoną parę liter. Dokładniej mówiąc generowano ciągi znaków typu „xx x”, gdzie x oznacza jakąś literę.




Rysunek 13: Przykład wygenerowanego "ab a". Tego typu sekwencje zostały użyte do tworzenia par liter.

Znak spacji oraz ostatnia litera zostaną wycięte. W ten sposób ostatnia litera przyjmie na siebie błąd generatora, a spacja zniweluje wpływ dodatkowego znaku na dwa pierwsze. Pozostaje wyciąć stworzoną parę.

Jeżeli jednak ostatnia litera miałaby zaburzyć wygląd dwóch pierwszych, to wpływ ten zostanie rozproszony poprzez wykorzystanie wszystkich 3 literowych (ze spacją w środku) kombinacji z powtórzeniami podczas generacji. Innymi słowy każda z poniższych możliwości zostanie wykorzystana podobną ilość razy, czyli docelowo około 6000.

"aa a", "aa b", …, "aa z",

"ab a", "ab b", …, "ab z"

… , … , …, ...

"zz a", "zz b", …, "zz z"

Kombinacje z powtórzeniami pożądanych ciągów znaków.

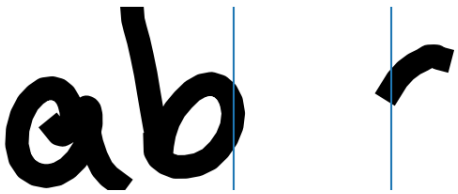
Generowanie danych jest kosztowne czasowo. Wygenerowanie 100 egzemplarzy każdej z 676 par z wykorzystaniem dwu wątkowego procesora zajmuje około 50dni. Problem ten będzie omówiony w rozdziale 3.6

### 3.3.2 Tworzenie i konwertowanie danych

Program generujący zbiór danych realizuje poniższe kroki wykonywane w pętli:

1. Wygenerowanie ciągu znaków. Zmiana generowanego ciągu co iterację zgodnie z zasadą wykorzystania wszystkich kombinacji pożądanych ciągów znaków (opisane na koniec poprzedniego rozdziału).
2. Pogrubienie liter i zapisanie zbioru zawierającego współrzędne punktów pociągnięć pióra jako stan interfejsu matplotlib.pyplot [17] .

   Wyboru grubości dokonywano na podstawie ostatecznego wyglądu pary liter (po wycięciu, rozmazaniu, pomniejszeniu itd.)



Rysunek 14: wygenerowane i odpowiednio pogrubione "ab c"

3. Znalezienie przerwy oddzielającej parę liter od nadmiarowego znaku.

   Zostało to zrealizowane poprzez znalezienie największej odległości pomiędzy sąsiadującymi ‘pociągnięciami pióra’ (takimi pośrodku których nie ma fragmentu innego ‘pociągnięcia pióra’) oraz które na siebie nie nachodzą. Przerwa oznaczona jest niebieskimi liniami na rysunku 14.

4. Wyznaczenie miejsca odcięcia pary liter w taki sposób, by jej umiejscowienie było centralne.

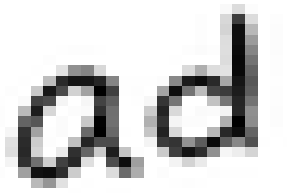
Rysunek 15: „ad” wycięte z wygenerowanego „ad a”

5. Zamiana obiektu matplotlib.pyplot [17] na obrazek
6. W celu dodania realizmu zastosowano rozmycie obrazka. Użyto algorytmu ImageFilter.GaussianBlur [18] i promienia rozmycia równego 6

   Wyboru promienia rozmycia dokonano na podstawie ostatecznego wyglądu pary liter.

Rysunek 16: rozmyte „ad” z rysunku 15

7. Próbkowanie obrazu w dół, do rozmiaru 28x28. Generowanie oraz przetwarzanie powstającego zbioru było na tyle kosztowne czasowo, że wybierając algorytm zmniejszający rozmiar nie skupiano się na jego złożoności, lecz na końcowej jakości. W celu pomniejszenia wygenerowanych obrazów użyto metody resize z biblioteki Pillow [19] używając algorytmu Image.LANCZOS. Wybrano ten algorytm, opierając się na opiniach znalezionych na forum [20][21] oraz samej dokumentacji biblioteki Pillow [22][19], gdzie algorytm opisany był jako filtr do próbkowania w dół wysokiej jakości („high-quality downsampling Lanczos filter”).



Rysunek 17: Zmniejszone "ad" z rysunku 16
. Powiększone w celu uwidocznienia szczegółów.

8. Następnie zapisywano obrazek jako czarno-biały (za pomocą Image.convert [23]), używając bezstratnego formatu PNG [24].

Otrzymaliśmy potrzebny format, czyli obrazek o rozmiarze 28x28 pikseli zawierający parę liter o odpowiedniej grubości.

### 3.3.3 Przetworzenie zbioru danych

Po stworzeniu zamierzonej ilości danych, czyli obrazków w formacie PNG, należało je przetworzyć tak, aby były możliwe do sprawnego odczytywania przez sieć. Biblioteka z której korzystano (PyTorch) ma gotowe rozwiązania pobierające i wczytujące najpopularniejsze zbiory danych (m.in. MNIST [13], EMNIST [14]). Zdecydowano się z tego skorzystać i przetworzyć posiadany zbiór danych w taki sposób, aby udawał zbiór MNIST. Następnie w celu użycia zbioru skorzystano z istniejących rozwiązań: torchvision.datasets.MNIST [25] oraz torch.utils.data.DataLoader [26].

Na stronie [13] zawarto opis w jaki sposób tworzone są pliki przechowujące zbiór MNIST. Nie stworzono jednak swojego programu, lecz użyto znalezionego konwertera potrafiącego zrealizować postawione wyżej zadanie [27]. Należało jedynie dostosować strukturę katalogów i można było z niego skorzystać.

### 3.3.4 Zmiany w bibliotece PyTorch

Zbiór został zapisany w formacie użytym w MNIST, jednak potrzebne były jeszcze minimalne zmiany w kodzie biblioteki PyTorch. Normalnie torchvision.datasets.MNIST pobiera (jeżeli potrzeba) zbiór MNIST a następnie procesuje go, czyli m.in. sprawdza poprawność danych oraz zapisuje w pożądanym formacie (rozszerzenie .pt). Procesowanie wykonywane jest tylko po uprzednim pobieraniu. W przypadku naszego zbioru należało wyłączyć pobieranie oraz uruchomić procesowanie (tylko podczas pierwszego uruchomienia sieci). W tym celu zmieniono klasę torchvision.datasets.MNIST.

Poprzednie zachowanie biblioteki przyniosło w pewnym momencie problemy (nadpisywanie zbioru stworzonego z par liter pobieranym zbiorem MNIST, niezrozumiałe zachowanie poprzez pozostawione przeprocesowane pliki .pt pochodzące z oryginalnego zbioru MNIST). W celu analizy tych problemów korzystano z odwrotnego konwertera (MNIST format na PNG) [28]. W ciągu dalszej pracy nad generatorem i siecią tworzono wiele zbiorów danych i czasem upewniano się, że do sieci głównej podpięto odpowiedni zbiór także dzięki odwrotnemu konwerterowi.

### 3.3.5 Podsumowanie pracy programu generującego

Tworzenie zbioru zapisać można w postaci kilku kroków:

Wykonywane w pętli:

1. Generowanie odpowiedniego ciągu znaków („xx x”).

   Co iterację inny ciąg zgodnie z zasadą wykorzystania wszystkich kombinacji (opisane na koniec poprzedniego rozdziału).

2. Przetwarzanie wygenerowanego ciągu znaków na pożądany format.
3. Zapis stworzonej pary.

Przetworzenie stworzonego zbioru do formatu MNIST.

## *3.4 Odrzucanie błędnych danych*

### 3.4.1 Poprawność generowanych danych

Wygenerowano pierwsze zbiory danych zgodnie z krokami przedstawionymi w rozdziale 3.3. Zauważono, że nie wszystkie wygenerowane elementy zbioru danych są stworzone poprawnie. W celu identyfikacji problemu stworzono zbiór zawierający 100 egzemplarzy każdej pary. Ręcznie sprawdzając wybrane spośród 676 możliwości pary liter zauważono, że:

- Pierwsza grupa par (np. ‘ab’, ‘ah’, ‘gi’, ‘pp’) została stworzona całkowicie, lub niemal całkowicie poprawnie (około 100% par poprawnych).
- Druga grupa par (np. ‘vj’, ‘xi’, ‘zx’) została wygenerowana niemal całkowicie źle. Wiąże się to najprawdopodobniej z rzadkością występowania niektórych par liter w zbiorze [29] na podstawie którego uczono generator [16].
- Trzecia grupa par stworzona była tylko w jakimś stopniu dobrze.

Sprawdzono w jakich przypadkach powstały złe egzemplarze danych i za każdym razem związane było to ze złym rezultatem zwróconym przez główny generator [16], czyli krok pierwszy z rozdziału ‘proces tworzenia zbioru’?. Nie zauważono aby dalsze kroki (wycinanie i przetwarzanie danych) spowodowały jakiekolwiek problemy.

Sieć główna nie była w stanie nauczyć się tworzyć nowych par liter na podstawie tak niedoskonałego zbioru, co nie było zaskoczeniem.

#### *3.4.2 Rodzaje błędów w generowanych danych*

Drugą grupę par można by ręcznie odrzucić nieznacznie zmniejszając zbiór, jednak sporą część danych stanowiła grupa trzecia. Pozbywając się wszystkich par należących do grupy drugiej i trzeciej drastycznie zmniejszylibyśmy możliwe pary liter.

Zauważono kilka problemów: Brak którejś litery, brak spacji, dodatkowa litera, bliżej nieokreślone wzory przypominające lub nie przypominające pożądanego zestawu znaków.


Rysunek 18: Przykładowe źle wygenerowane pary znaków, przed wycięciem.
1. Nic nie przypominający wzór, 2. Źle wygenerowane ‘ez s’,
3. ‘ayn’ bez spacji, 4. ‘gz a’, brakuje litery ‘z’ i spacji.

### *3.4.3 Odrzucenie błędnych danych*

Podjęto próbę odfiltrowania kilku z wymienionych wyżej problemów – wszystkich które nie przypominają pożądanego zestawu liter „xx x”. Weryfikując rozmiary poprawnie oraz błędnie wygenerowanych danych wybrano minimalną poprawną szerokość tworzonej pary liter na 30% całości obrazka oraz przerwę pomiędzy parą a dodatkowym znakiem na 19% obrazka. Do programu generującego dane (rozdział 3.3) dodano krok sprawdzający, czy wyznaczona para liter oraz spacja spełniają wymagania co do rozmiaru, w ten sposób w następnie tworzonych zbiorach odrzucano niepoprawne przypadki.

W przykładowym zbiorze zawierającym po 100 par liter odrzucono:

- Łącznie 7062 egzemplarzy (10.45% wszystkich).
- Egzemplarze należące do 659 różnych par (97.49%).
- Najczęściej odrzucane pary zaczynały się na ‘z’, łącznie 807 odrzuconych (31% wszystkich par na ‘z’)
- Najczęściej odrzucana para, to ‘zg’ - 59 razy (59% wszystkich ‘zg’)

Po przejrzeniu powstałego zbioru zauważono znaczną poprawę – dużo trudniej znaleźć błędy typu 1, 3, 4 z rysunku 18, czyli te które mieliśmy odrzucić oraz minimalna ilość z pośród odrzuconych egzemplarzy była poprawna.

Ponownie spróbowano uruchomić sieć główną, jednak nie była ona w stanie nauczyć się tworzyć nowych par liter. Postanowiono podjąć dalsze kroki w odrzucaniu błędnie wygenerowanych danych, których wciąż było bardzo dużo.

Algorytm można byłoby poprawić np. zaostrzając warunki minimalnych szerokości dzięki wyznaczeniu ich dla każdej pary z osobna, lub biorąc pod uwagę przeciętne szerokości używanych liter. Nie zdecydowano się jednak na to, ponieważ problemy które mieliśmy odfiltrować (typ 1,3,4 z rysunku 18) znacząco się zmniejszyły, a po dokładniejszym przejrzeniu danych zauważono, że największym problemem jest błąd typu 2 z rysunku 18.

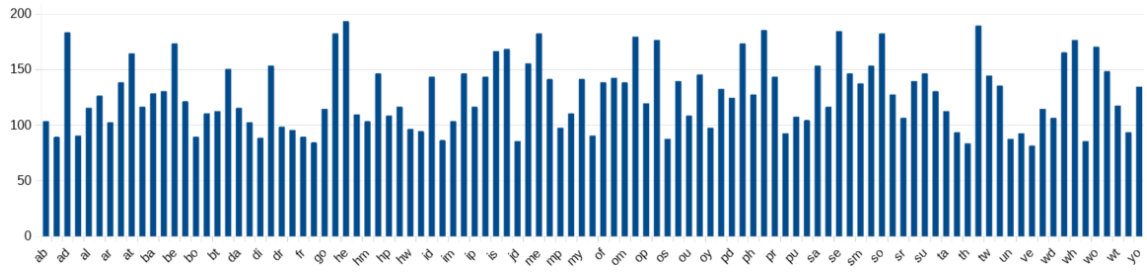
## *3.5 Odrzucanie błędnych danych – sieć neuronowa*

Do odfiltrowania błędów typu 2 z rysunku 18 bardzo dobrze nadają się algorytmy rozpoznające pismo odręczne (ang. handwritten text recognition, HTR). Wyszukano program HTR wykorzystujący sieć neuronową do rozpoznawania tekstu [30]. Wraz z programem pobierany był model wytrenowany na zbiorze IAM [29], czyli tym samym którego używał generator. Ominięto w ten sposób potrzebę uczenia programu od zera. Po wczytaniu wytrenowanego modelu otrzymaliśmy gotowy silnik przystosowany do rozpoznawania tekstu tworzonego przez generator. Nazywać go będziemy OCR, tak jak klasa programów do której należy - optycznego rozpoznawania znaków (ang. optical character recognition, OCR).

Sieć rozpoznająca nie okazała się mieć bardzo dobrych rezultatów. Autor twierdził o 74% skuteczności na zbiorze IAM, używając dodatkowo algorytmu „vanilla beam search decoding" dopasowującego rozpoznany tekst do rzeczywiście istniejących możliwości. (cyt. „74% of the words from the IAM dataset are correctly recognized by the NN when using vanilla beam search decoding.") [30]. Warto nadmienić, że autor osiągnął jeszcze lepsze rezultaty używając „word beam search decoding".

W naszym przypadku rozpoznawanie jest zdecydowanie gorsze. Dokonano testu generując 200 egzemplarzy każdej pary liter. Poniższe statystyki dotyczą podwójnego filtrowania – najpierw za pomocą programu z rozdziału , a następnie OCR:

- Blisko połowy par w całości nie rozpoznano (48.67%)
- Pary będące jednocześnie wyrazami rozpoznano z bardzo wysoką skutecznością: (ad 91,5%, he 96.5 %, it 84%, me 91%)
- Jest 103 pary w których poprawnie rozpoznano ponad 40% wyrazów
- Najlepsze 103 pary rozpoznano z dość dobrą, bo 63.56% skutecznością




Rysunek 19: Wykres słupkowy obrazujący w jaki sposób rozkładają się ilości najlepiej rozpoznanych par.

Wyniki nie są bardzo zaskakujące biorąc pod uwagę, że:

- Podane wyżej statystyki dotyczą użycia dwóch algorytmów – OCR oraz programu z rozdziału ***3.4.1***, który dodatkowo obniża ilość rozpoznanych egzemplarzy.
- Rozpoznawano pary liter, natomiast sieć uczona była rozpoznawania całych wyrazów. Dla par liter będących wyrazami osiągnięto bardzo dobre rezultaty.
- Autor osiągnął 74% używając programu „vanilla beam search decoding" [30] dopasowującego rozpoznane wyrazy do rzeczywiście istniejących. Jednak w przypadku par liter nie ma to sensu, gdyż wszystkie pary rzeczywiście istnieją.
- W zbiorze danych istnieje zauważalna część niepoprawnych wyrazów z błędem typu 2 z rysunku 18.

Zdecydowano się zrezygnować ze źle rozpoznawanych par liter. Analizując wyniki wymienionego wcześniej testu na 200 egzemplarzach obliczono, że: chcąc otrzymać zbiór 103 par liter zawierających po 200 dobrze wygenerowanych egzemplarzy należy łącznie stworzyć około 34314 egzemplarzy. Stanowiłoby to 167% wielkości podstawowego zbioru.

Niesie to ograniczenia możliwości sieci głównej, ale ma też pozytywne aspekty. Zmniejsza czas potrzebny na wygenerowanie zbioru danych około czterokrotnie (103 z 676 par, ale potrzeba wygenerować dodatkowe 67% danych). Jednocześnie wciąż będziemy mogli stworzyć program generujący realistycznie wyglądające pary znaków, który najprawdopodobniej poradziłby sobie także ze zbiorem pełnej wielkości. Ewentualnie należałoby np. powiększyć sieć lub po prostu stworzyć 7 instancji - każda zajmująca się inną częścią zbioru par.

Ostatecznie użyto progu 43% który odpowiadał generowaniu 97 najlepszych par liter.

## *3.6 Generowanie całego zbioru*

### 3.6.1 Kroki programu generującego

Kroki programu generującego:

1. W pętli wykonywano generowanie danych oraz zautomatyzowany proces odrzucania par ze słabą skutecznością, czyli:
   - Wygenerowanie pewnej ilości danych (zgodnie z rozdziałami 3.3 oraz 3.4). Z każdym krokiem pętli ilość rosła dwukrotnie, poza ostatnim krokiem, w którym tworzono brakującą ilość.
   - Sprawdzenie poprawności wygenerowanych danych za pomocą programu OCR z rozdziału 3.5. Usunięcie tych niepoprawnych (początkowo przenoszono je do innego katalogu celem analizowania przyczyny).
   - Użycie programu filesCounter.py tworzącego plik z podliczeniem ilości wygenerowanych egzemplarzy dla każdej z par.
   - Odrzucenie z procesu par ze słabą skutecznością generacji.
   - Powtórz od początku, lub przejdź do kroku 2 jeżeli wygenerowano odpowiednią ilość.

2. Wygenerowanie brakujących danych - na podstawie aktualnie istniejącej ilości danych szacowano skuteczność dla konkretnej pary i wołano proces tworzący odpowiednią ilość.

3. Użycie programu renameFiles.py, czyli zapewnienie ciągłości w nazwach plików.

4. Przetworzenie do formatu MNIST (rozdział 3.3.3 Przetworzenie zbioru danych)

Program dałoby się wykonać bez komunikacji poprzez pliki. Nie zdecydowano się na to jednak, ponieważ zastosowany sposób nie powinien mieć zauważalnego wpływu na złożoność czasową oraz oddawał wcześniej wypróbowaną metodę tj. ręcznego wołania programów krok po kroku.

### 3.6.2 Problem z mocą obliczeniową

Jak wspominano pod koniec rozdziału generowanie zbioru było kosztowne czasowo już na tamtym etapie. Obliczono wtedy zapotrzebowanie na 50 dni ciągłej pracy dwurdzeniowego komputera, nie biorąc pod uwagę także bardzo czasochłonnego procesu nauki sieci głównej. W związku z rezygnacją z błędnie tworzonych par liter ilość danych do wygenerowania zmniejszyła się około czterokrotnie. Uwzględniając jednak wiele dodatkowych operacji opisanych w rozdziale czas generowania zmniejszył się 2.5 krotnie – około 20 dni dla zamierzonego zbioru.

### 3.6.3 Wieloprocesowość

Pojedynczy proces nie używał jednak całej dostępnej mocy obliczeniowej. Podczas generacji średnie użycie dwurdzeniowego procesora wynosiło zaledwie około 30 %. Zdecydowano się na użycie wielo-procesowości. Użyto w tym celu obiektu Pool z biblioteki multiprocessing (język Python) [31]. Należało wykorzystać całą moc obliczeniową uruchamiając przy tym minimalną ilość procesów, tak aby uniknąć strat związanych ze zbyt dużym obciążeniem planisty. Sprawdzono, że odpowiednią liczbą byłyby 4 równoległe procesy, dla których czas generacji powinien wynosić trochę powyżej 5 dni.

### 3.6.4 Obliczenia w chmurze

Niestety nie dysponowano komputerem mogącym pracować bez przerwy, co powodowałoby znaczne wydłużenie czasu jak i problemy z integracją danych.

W pierwszej kolejności zarejestrowano się na portalu PLGrid [32], w celu skorzystania z mocy obliczeniowej jednego z superkomputerów [33]. Skupiono się na wykorzystaniu Zeusa [34] i zdawało się, że będzie wspierał wszystkie potrzebne technologie. Zapoznano się ze sposobem w jaki należy korzystać z superkomputera (system kolejkowania, załączanie modułów), jednak okazało się, że uruchomienie programu nie będzie możliwe. Problemem był brak wsparcia dla biblioteki TensorFlow, pomimo możliwości załadowania takowej.

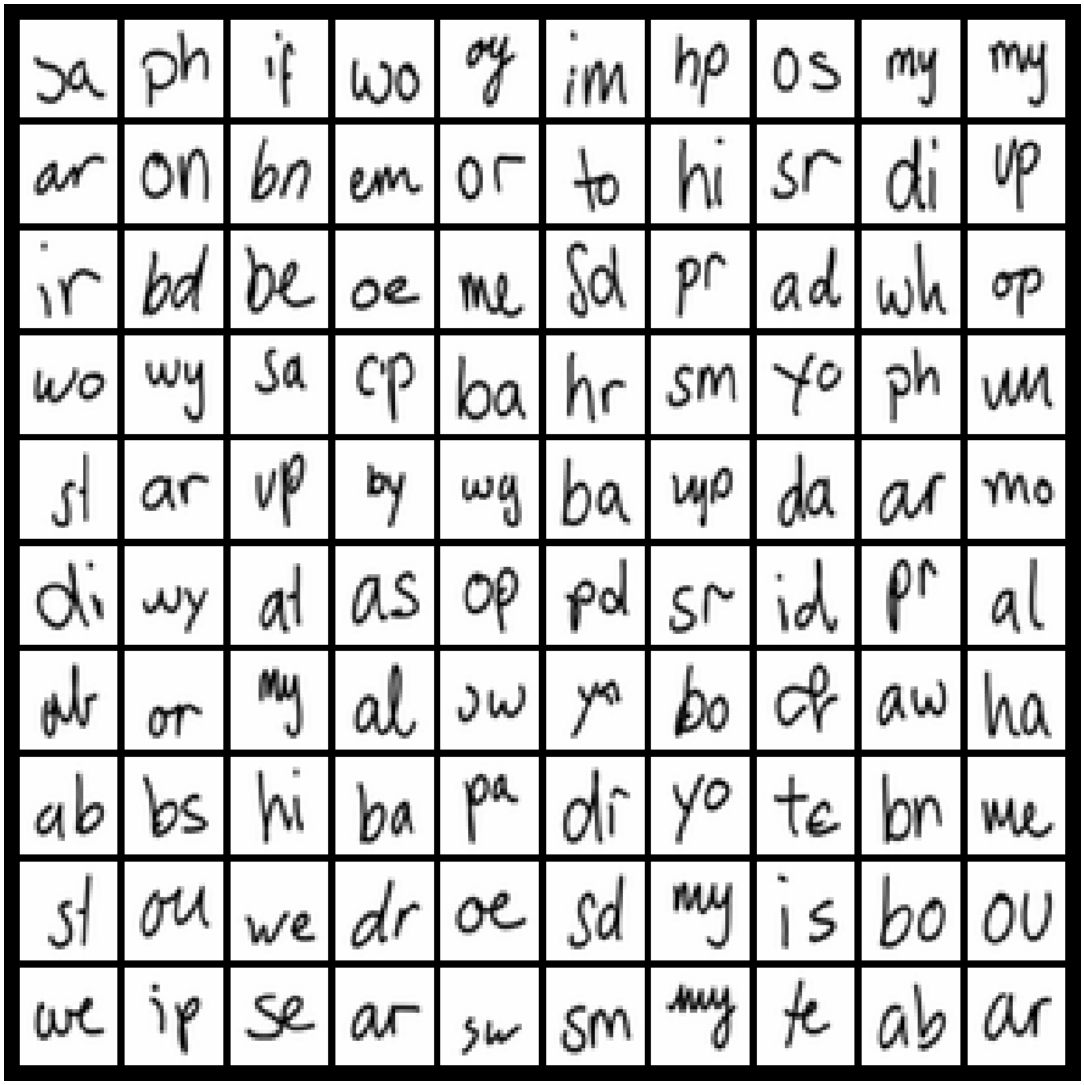
W związku z powyższym zdecydowano się na użycie chmury obliczeniowej Google Cloud Platform. Skorzystano w tym celu z programu GCP Free Tier, oferującego darmowy dostęp do części usług platformy z limitem 300$ do wykorzystania.

Plarforma GCP posiadała w swojej ofercie bardzo wydajne karty graficzne. Wykorzystanie takiego rozwiązania miało sens, gdyż generator [16] zaimplementowany był z użyciem sieci neuronowej i biblioteki tensorflow [35]. Niestety użycie odpowiedniej maszyny wirtualnej okazało się nieopłacalne. Karta graficzna z której korzystanie było drogie, wybudzana była tylko chwilami, a zdecydowaną większość czasu obliczeniami zajmowały się procesory, nawet w konfiguracjach które zawierały ich wiele.

Zrezygnowano z karty graficznej. Wykonano próby maszyn z różną ilością procesorów. Dla dużej ilości (przynajmniej 16) wąskim gardłem był dysk. Porównywano rozwiązania zawierające po kilka procesorów i najbardziej wydajną okazała się konfiguracja n1-standard-2 (2 procesory, 7.5 GB RAM, dysk SSD), dla której odpowiednie było 5 równoległych procesów (zgodnie z zasadami z rozdziału 3.6.3 Wieloprocesowość). Była ona wystarczająca, gdyż tworzenie całego zbioru zająć miało około 5 dni.

Celem zaoszczędzenia czasu i szybszego przetestowania w pierwszej kolejności powstał zbiór zawierający po 3000 egzemplarzy każdej pary. Równolegle podczas pracy z pierwszym zestawem danych tworzono następne 3000 egzemplarzy, które później połączono w jeden zbiór.

Ostatecznie dane prezentowały się w taki sposób:



Rysunek 20: Dane wejściowe używane do uczenia sieci głównej.

# Rozdział 4 Architektura i uczenie modelu

## *4.1 Architektura sieci*

W pierwszej kolejności zdecydowano się na zaimplementowanie modelu działającego dla cyfr. Implementacja taka w założeniu stanowić miała punkt zaczepny dla dalszej pracy, być może pomagając przy testowaniu tworzonych zbiorów danych. Skorzystano z modelu i parametryzacji GAN zaproponowanego tutaj [36]. Bazuje on na historycznie pierwszej pracy, która wprowadziła generatywne sieci przeciwstawne [1].

Tworzona jest sieć neuronowa o architekturze wielowarstwowego perceptronu (ang. multilayer perceptron), czyli taka jaką polecali autorzy GAN [1]. Każda warstwa gęsto połączona z następną, bez sprzężeń zwrotnych (FNN) oraz posiadająca więcej niż 1 warstwę ukrytą (DNN) (skróty z rozdziału 1.5).

Architektura sieci generatora:

Warstwa wejściowa: 100 neuronów

funkcja aktywacji LeakyReLU(0.2)

Pierwsza warstwa ukryta: 256 neuronów

funkcja aktywacji LeakyReLU(0.2)

Druga warstwa ukryta: 512 neuronów

funkcja aktywacji LeakyReLU(0.2)

Trzecia warstwa ukryta: 1024 neuronów

funkcja aktywacji tangens hiperboliczny

Warstwa wyjściowa: 784 (rozmiar obrazka 28x28) neuronów

Architektura sieci dyskryminatora:

Warstwa wejściowa: 784 neurony (rozmiar obrazka 28x28)

funkcja aktywacji LeakyReLU(0.2)

Technika regularyzacji DropOut(0.3)

Pierwsza warstwa ukryta: 1024 neuronów

funkcja aktywacji LeakyReLU(0.2)

Technika regularyzacji DropOut(0.3)

Druga warstwa ukryta: 512 neuronów

funkcja aktywacji Leaky ReLU(0.2)

Technika regularyzacji DropOut(0.3)

Trzecia warstwa ukryta: 256 neuronów

sigmoidalna funkcja aktywacji

Warstwa wyjściowa: 1 neuron

## Transformacje liniowe:

Wszystkie warstwy połączone są transformacjami liniowymi:

$$y = xA^T + b$$

## Leaky ReLU:

Wszystkie warstwy oprócz wyjściowych używają funkcji aktywacji Leaky ReLU, tak jak polecono w pracy [37]. Pojęcie opisane zostało w rozdziale 1.3.

## DropOut:

Wszystkie warstwy dyskryminatora oprócz wyjściowej jako techniki regularyzacji używają odpadania (opisane w rozdziale 1.6.4 Technika regularyzacji - odpadanie) tak jak poleca się robić w sieciach GAN [1]. Ma to zapobiegać jego przetrenowaniu (1.6.3 Przeuczenie).

## Dane wejściowe:

Dane wejściowe dyskryminatora to czarno-biały obrazek zapisany jako wektor z wartościami znormalizowanymi z [0,255] do [-1,1].

Dane wejściowe generatora to wektor 100 losowych wartości. W celu optymalizacji używane są wartości losowe z rozkładem normalnym [37], skorzystano z metody randn [38].

## Dane wyjściowe:

Ostatnia warstwa dyskryminatora powinna zwracać prawdopodobieństwo z przedziału [0,1]. Dlatego jest to 1 neuron, a sigmoidalna funkcja aktywacji zapewnia przedział wartości [0,1].

Chcemy aby generator tworzył obrazki zgodne z tymi które przyjmuje dyskryminator, dlatego:

- warstwa wyjściowa generatora ma tyle samo neuronów co warstwa wejściowa dyskryminatora
- generator dla warstwy wyjściowej używa funkcji aktywacji tangens hiperboliczny zwracającej zakres wartości [-1,1].

## Podobieństwo sieci:

Sieć neuronowa generatora przypomina sieć dyskryminatora, tylko z odwrotnym przepływem danych, dlatego że zależy nam na wyrównanej nauce obydwu sieci.

## *4.2 Uczenie modelu*

### 4.2.1 Teoretyczny opis procesu uczenia

#### 4.2.1.1 Sposób działania

Działanie dyskryminatora:

- Przyjmuje on obrazek w postaci wektora i zwraca prawdopodobieństwo, że jest on parą znaków.

Działanie generatora:

- Przyjmuje wektor 100 losowych (z rozkładem normalnym) wartości i na ich podstawie zwraca obrazek również w postaci wektora.

#### 4.2.1.2 Epoka, seria i mini-grupa

Ze wszystkich danych tworzono 5820 serii/mini-grup (ang. batch) po 100 egzemplarzy. Co każdą taką serię obliczana była łączna funkcja straty i wykonywane było trenowanie modelu. Przetrenowanie wszystkich danych, czyli 5820 serii nazywane jest epoką. Sieć musi uczyć się wiele epok (ang. epoch), aby zwracane wyniki były zadowalające.

#### 4.2.1.2 Proces uczenia

Proces uczenia wykonywany w pętli:

1. Stwórz serię danych z prawdziwych obrazków.

   Stwórz serię danych z obrazków zwróconych przez generator.
2. Oblicz funkcję straty dla dyskryminatora, jako sumę funkcji straty dla pierwszych i drugich danych.
3. Wytrenuj dyskryminator korzystając z optymalizatora Adam.
4. Stwórz serię danych za pomocą generatora.
5. Oblicz funkcję straty używając dyskryminatora.
6. Wytrenuj generator korzystając z optymalizatora Adam.
7. Powtórz od początku

#### 4.2.1.3 Problem z uczeniem generatora

We wprowadzeniu teoretycznym generatywnych sieci przeciwstawnych stwierdzono, że celem nauki generatora będzie minimalizowanie (rozdział 1.8.2)

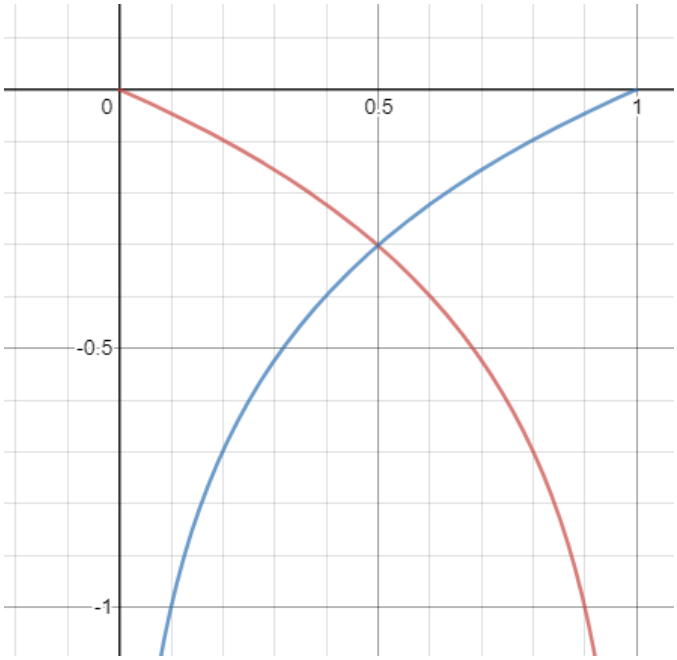
$$\log\left(1 - D\left(G\left(z\right)\right)\right)$$

W praktyce często się takiego rozwiązania nie stosuje. Związane jest to z tym, że im gorzej dyskryminator oceni dane wygenerowane przed generator, tym wartość zwrócona

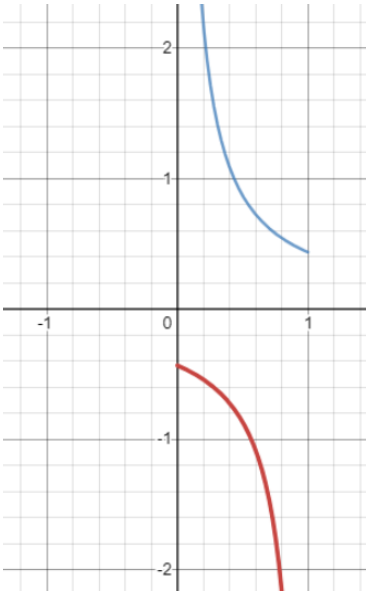
przez gradient będzie słabsza – wagi będą zmieniane coraz słabiej. Bardziej odpowiednia zdaje się być następująca zasada: im gorzej dyskryminator oceni generator, tym większe zmiany wag są potrzebne. Dlatego często minimalizowanie powyższego równania zamieniane jest na maksymalizowanie

$$\log\left(D\left(G\left(z\right)\right)\right)$$

Problem zobrazowano na dwóch poniższych wykresach:




**Rysunek 21 Funkcje: czerwona log(1-x), niebieska log(x), użyto [47]**




**Rysunek 22 Funkcje: czerwona d/dx(log(1-x), niebieska d/dx(log(x)), użyto [47]**

#### 4.2.1.4 Funkcja straty

Jako funkcji straty użyto binarnej entropii krzyżowej (ang. binary cross-entropy), opisana jest ona w rozdziale 1.6.6 Binarna entropia krzyżowa.

Nie korzystano z funkcji straty obliczonej dla każdego egzemplarza danych, lecz dla całej serii/mini-grupy (ang. mini-batch). Wykonywano to obliczając funkcję straty za pomocą średniej:

$$L = \frac{1}{rozmiarSerii} \cdot \sum_{n=0}^{rozmiarSerii} -([y_n \cdot \log(x_n) + (1 - y_n) \cdot \log(1 - x_n)])$$

gdzie rozmiarSerii = 100 (rozdział powyżej),

y oznacza wartość oczekiwaną, a x wartość podaną.

Strata dyskryminatora:

Strata dyskryminatora będzie sumą strat dla prawdziwych obrazków, z wartością oczekiwaną 1 (podstawiamy $y_n$=1 oraz $x_n$=D($x_n$)) oraz tych wygenerowanych przez generator z wartością oczekiwaną 0 (podstawiamy $y_n$=0 oraz $x_n$=D(G($z_n$))).

Prawdziwe obrazki: $$\frac{1}{rozmiarSerii} \cdot \sum_{n=0}^{rozmiarSerii} -\log(D(x_n))$$

Wygenerowane obrazki: $$\frac{1}{rozmiarSerii} \cdot \sum_{n=0}^{rozmiarSerii} -\log(1 - D(G(z_n)))$$

Strata: $$L = \frac{1}{rozmiarSerii} \cdot \sum_{n=0}^{rozmiarSerii} -[\log(D(x_n)) + \log(1 - D(G(z_n)))]$$

Strata generatora:

Strata generatora będzie stratą zwróconą przez dyskryminator dla wygenerowanych obrazków, tym razem wartością oczekiwaną jest 1 (podstawiamy $y_n$=1 oraz $x_n$=D(G($z_n$))).)

Strata: $$L = \frac{1}{rozmiarSerii} \cdot \sum_{n=0}^{rozmiarSerii} -[\log(D(G(z_n)))]$$

#### 4.2.1.5 Strata zgodna z teorią

Biblioteki z których korzystać będzie model nastawione są na minimalizowanie funkcji straty.

Zauważyć można, że dla generatora binarna entropia krzyżowa pokrywa się z teoretycznymi aspektami uczenia generatywnych sieci przeciwstawnych (rozdział 1.8.2).

Natomiast funkcja straty generatora wydaje się być nieodpowiednia. W rzeczywistości jest taka jak należy - wystarczy wziąć pod uwagę rozumowanie z poprzedniego rozdziału (rozdział 4.2.1.3 Problem z uczeniem generatora) oraz to, że maksymalizowanie jakiejś wartości odpowiada minimalizowaniu jej przeciwieństwa.

#### 4.2.1.6 Algorytm optymalizacyjny

Do uczenia sieci użyto algorytmu optymalizacyjnego Adam [39] (polecany do sieci GAN [37]). Optymalizacja wag następowała co każdą mini-grupę.
Użyto następujących współczynników:
Współczynnik uczenia $\alpha = 0.0002$,
Reszta użytych współczynników taka jaką polecali twórcy algorytmu [39] ($\beta_1 = 0.9$, $\beta_1 = 0.999$ and $\varepsilon = 10^{-8}$).

### 4.2.2 Praktyczny opis procesu uczenia

#### 4.2.2.1 Wybór środowiska

Trenowanie sieci było kosztowne obliczeniowo, 100 epok przekładało się na 23.5 godziny pracy komputera osobistego. W pierwszej kolejności uruchomiono maszynę wirtualną z kartą graficzną na Google Cloud Platform (GCP). Była to konfiguracja n1-standard-4 (4 procesory, 15 GB pamięci RAM, dysk SSD) wraz z kartą graficzną NVIDIA Tesla K80. Sprawdzono, że karta graficzna używana jest w znaczącym stopniu. Tym razem posiadano już komputer osobisty mogący pracować bez przerwy. Ostatecznie zdecydowano się na jego użycie z kilku powodów: czas obliczeń możliwy do przyjęcia, oszczędzanie darmowych środków na GCP, wygoda lokalnych obliczeń.

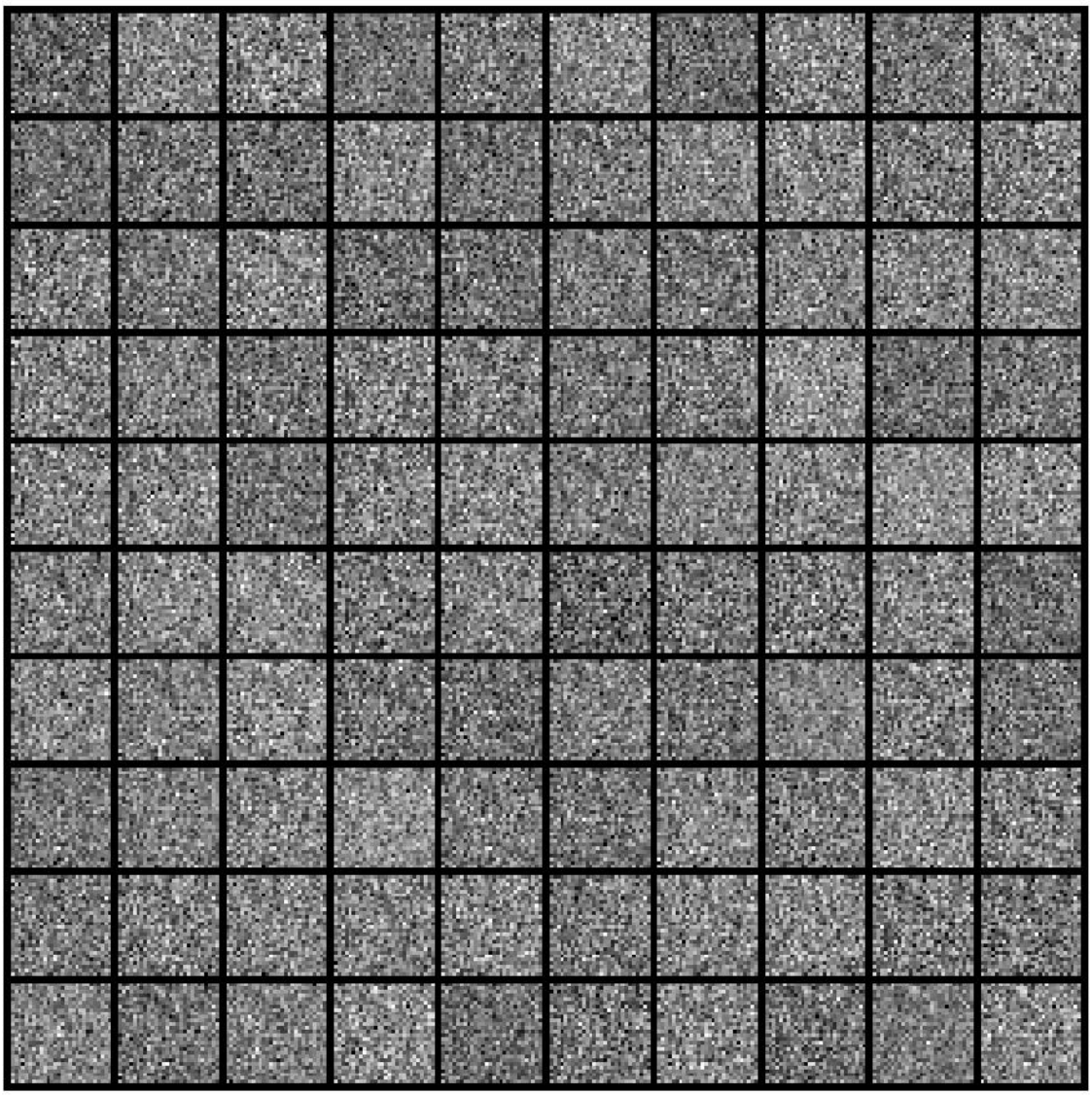
#### 4.2.2.2 Problem z przepełnianiem RAM

Model zaimplementowany został w bibliotece PyTorch [40], korzystano także z narzędzia Jupyter Notebook [41]. Napotkano jeszcze na problem związany z samym środowiskiem. Program nie zwalniał pamięci ram i po około 30 epokach kernel Jupyter Notebooka umierał (program zużywał wtedy ok. 14GB RAM). Sporą część uczenia sieci wykonano co jakiś czas manualnie restartując proces. Później problem rozwiązano korzystając z polecenia jupyter nbconvert. Ograniczono działanie sieci do 8 epok oraz automatycznie zapisywano i wznawiano pracę sieci z odpowiedniego stanu (korzystając z torch.save i torch.load [42]) . Z konsoli wołano listę komend, które wykonywały się jedna po drugiej.

#### 4.2.2.3 Przegląd procesu uczenia

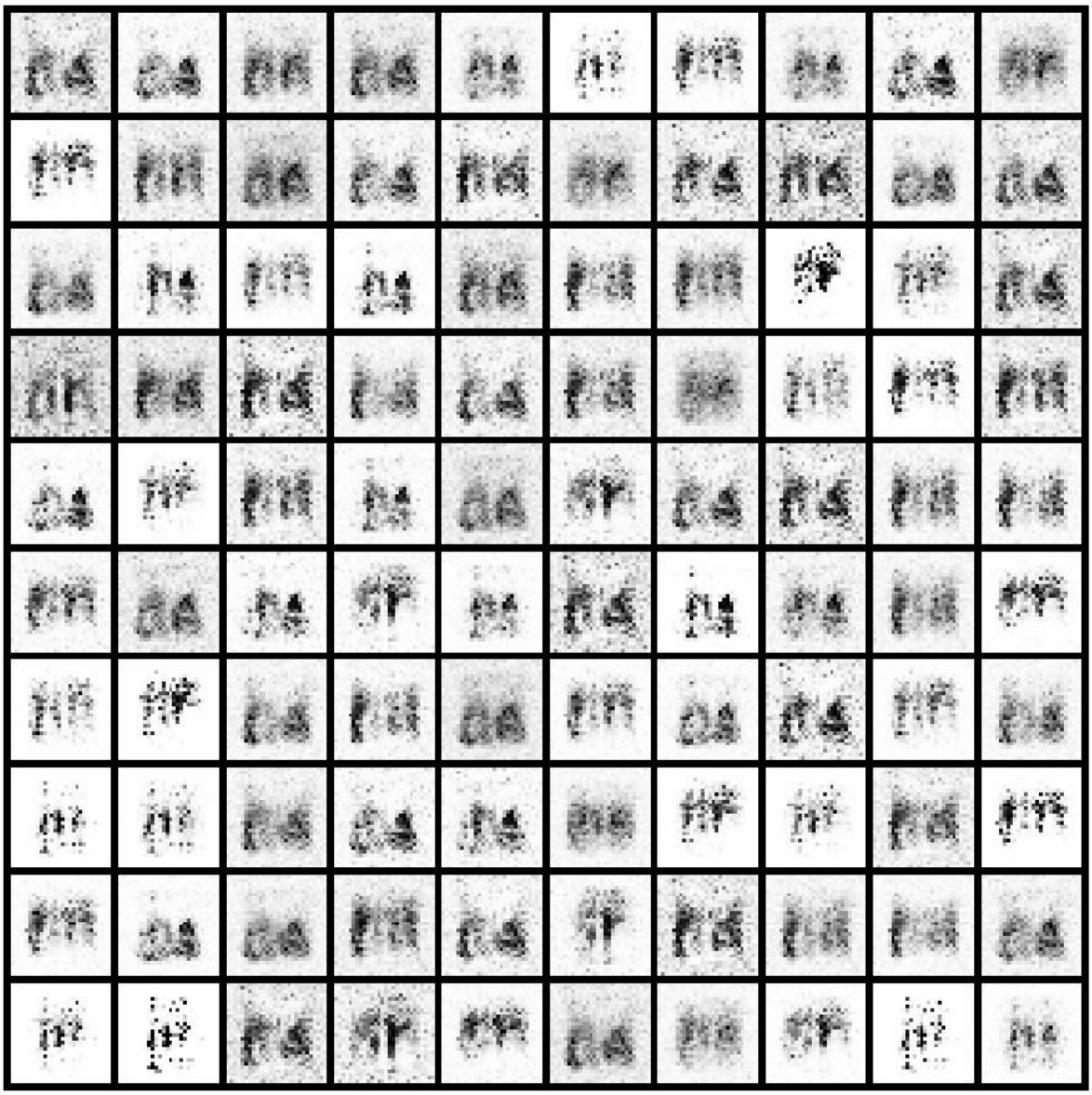
Przedstawiona poniżej seria obrazków zawierająca po 100 wygenerowanych egzemplarzy uwidacznia przebieg uczenia sieci.

Dane obrazujące ten proces generowane były dla takich samych wektorów wejściowych. Niesie to ze sobą taki efekt, że wybrany obrazek uwidacznia dane wygenerowane dla konkretnego wektora przez cały proces uczenia, co obserwować można jako udoskonalanie wyglądu konkretnej pary. Pozwala to prościej oceniać postępy sieci. Dzieje się tak tylko przez jakiś czas, ponieważ na skutek procesu uczenia konkretny wektor wejściowy może być przetwarzany na inną parę liter. Efekt ten będzie praktycznie niewidoczny na poniższych przykładach, ponieważ pomiędzy każdym z nich sieć robiła znaczące postępy i pary liter generowane dla poszczególnych wektorów zmieniały się.



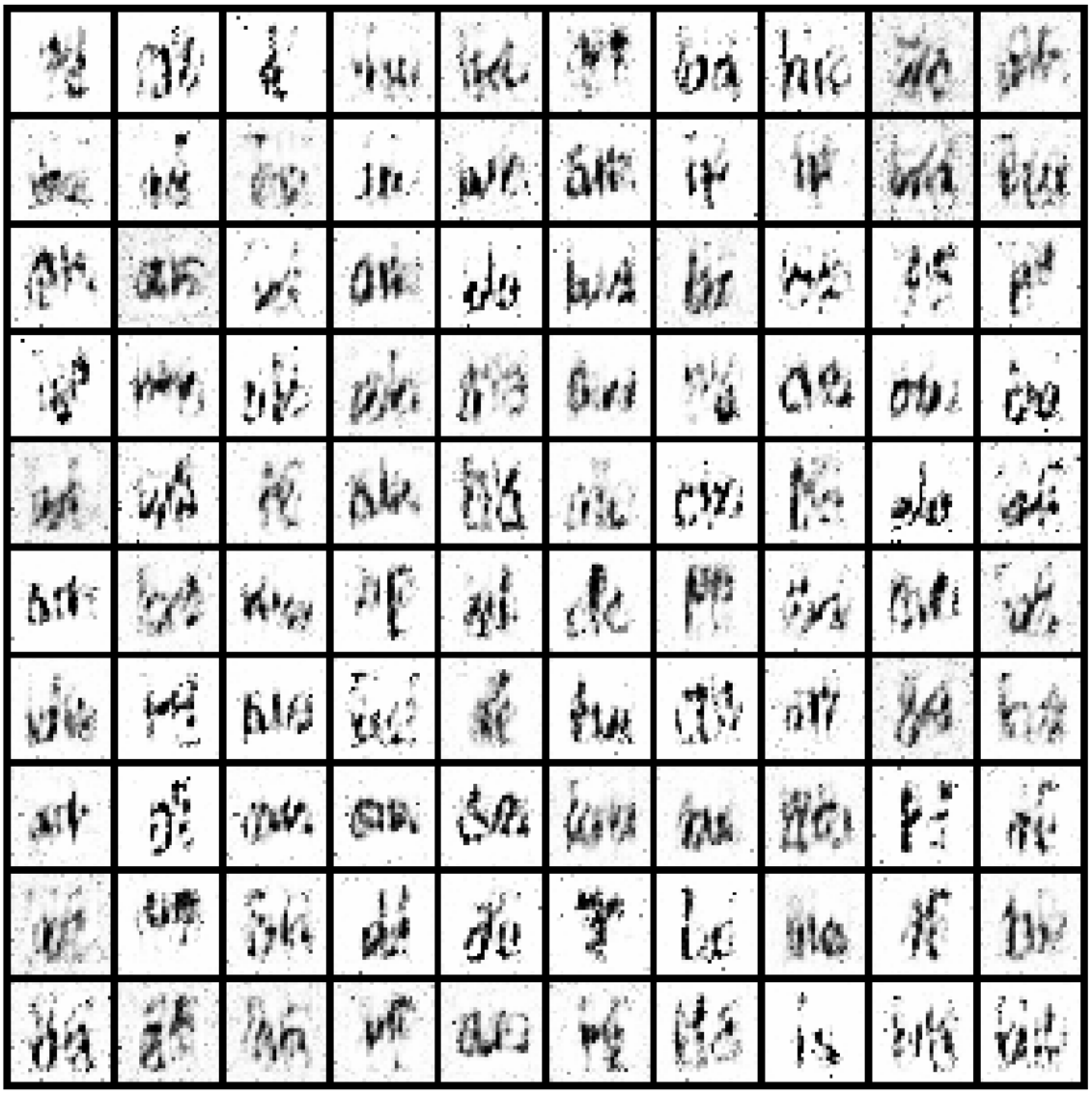
Rysunek 23: Pierwsze wyniki zwrócone przez generator (epoka 0 seria 100).

Początkowe wyniki wyglądają jakby generator zwrócił losowe wartości dla każdego piksela. Jedyną obserwacją jest zmiana odcienia pomiędzy poszczególnymi egzemplarzami.



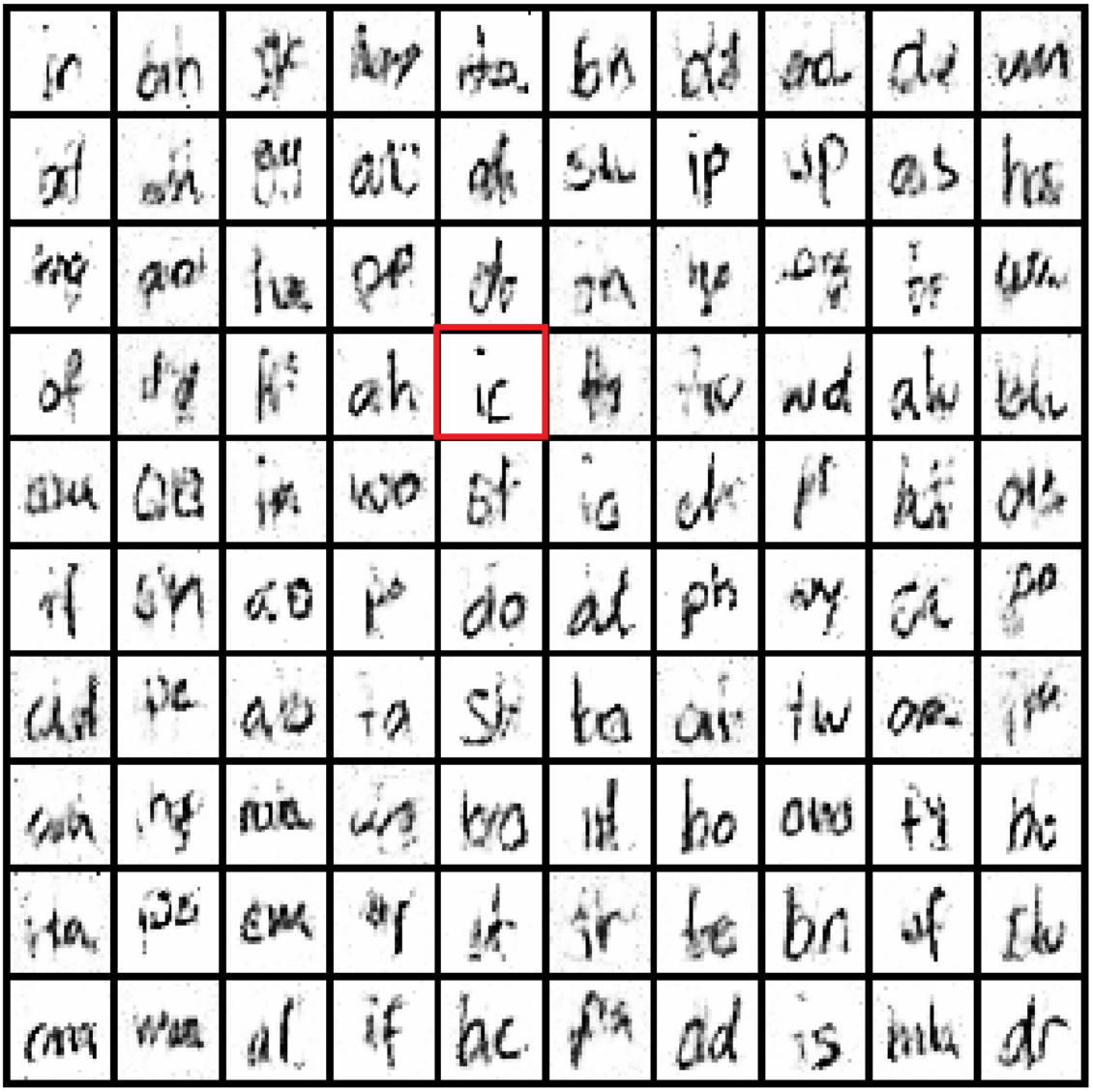
Rysunek 24: Wyniki generatora pod koniec zerowej epoki (epoka 0, seria 5800)

Generowane dane zaczynają nabierać odpowiednich kształtów. Ciemne piksele kumulują się w miejscach gdzie realnie zapisane są pary liter. Pomiędzy poszczególnymi egzemplarzami zauważyć można duże podobieństwa.



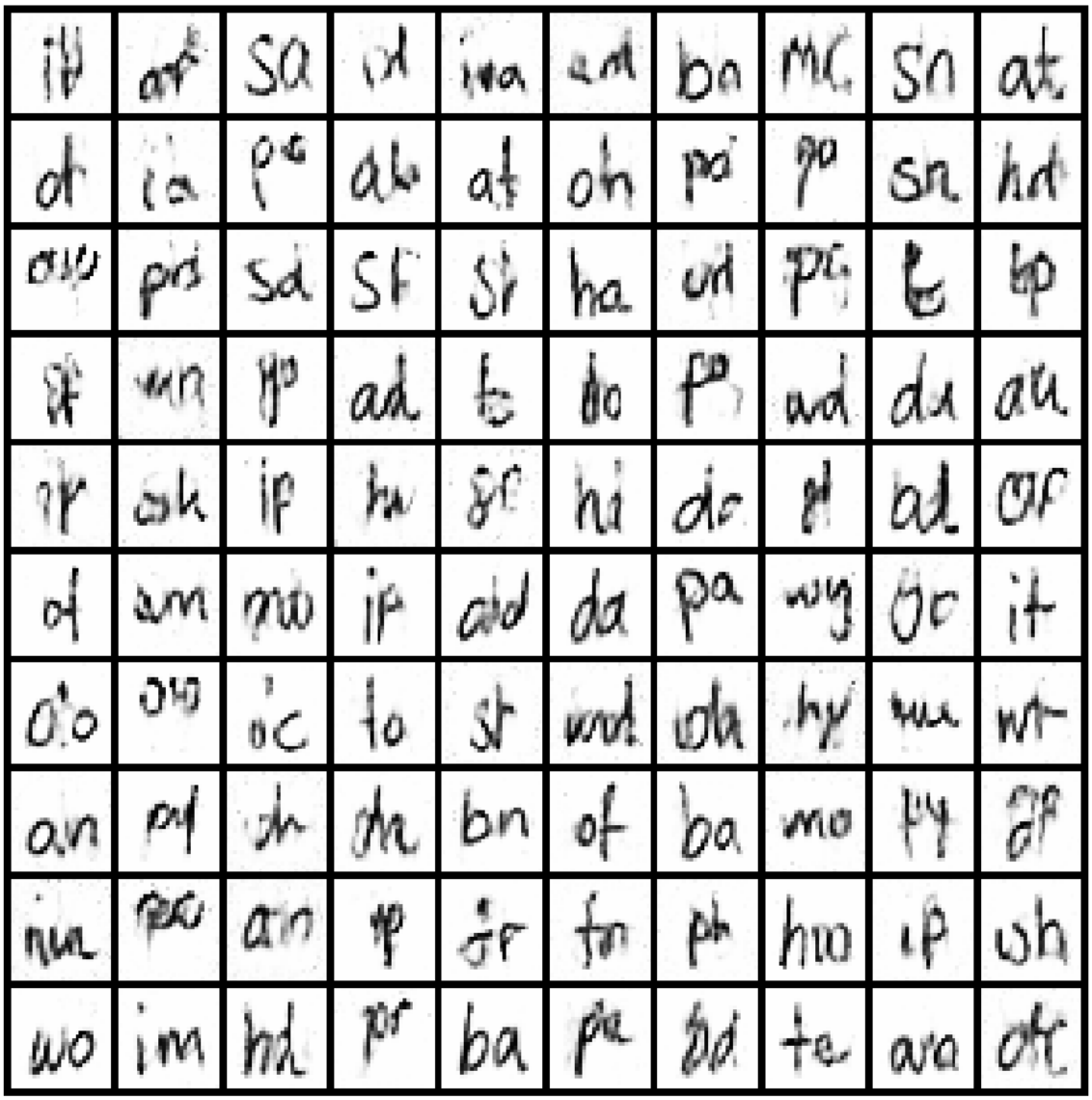
Rysunek 25: Wyniki generatora pod koniec piątej epoki (epoka 5, seria 5800)

Dane nabierają coraz bardziej realistycznego kształtu i zwiększyła się ich różnorodność. Niektóre egzemplarze daje się już rozpoznać, ale jakość pozostawia jeszcze wiele do życzenia.



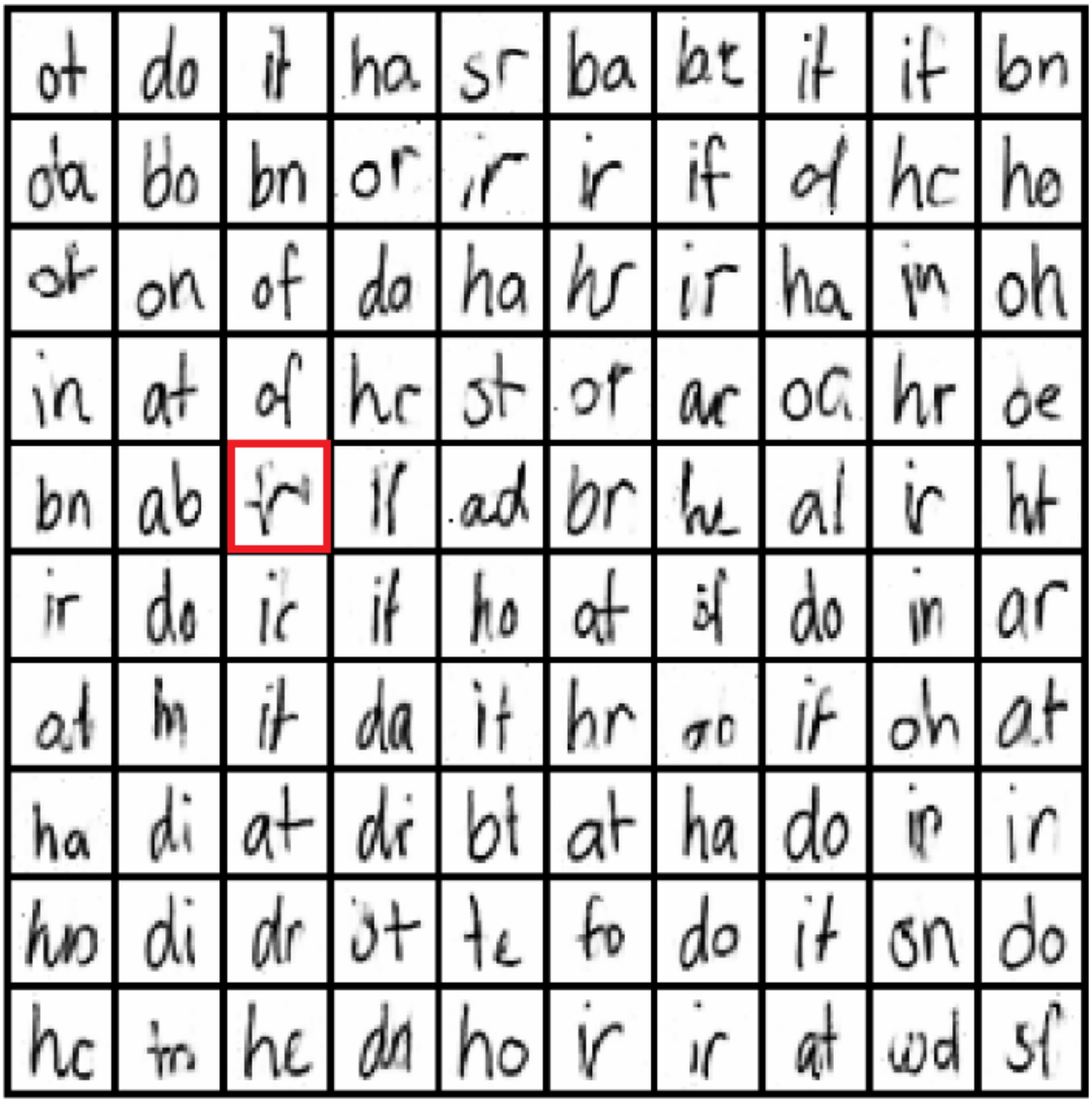
Rysunek 26: Wyniki generatora pod koniec dwudziestej piątej epoki (epoka 25, seria 5800)

Pary nie są już tak poszarpane i znacząco zmniejszyła się ilość pikseli rozrzuconych po całym obrazku. Sporą część da się już rozpoznać, niektóre egzemplarze wyglądają całkiem realistycznie (np. zaznaczone na czerwono ‘ic’).



Rysunek 27: Wyniki generatora pod koniec setnej epoki (epoka 100, seria 5800)

Coraz więcej par liter przypomina stworzone przez człowieka.



Rysunek 28: Wyniki generatora pod koniec tysięcznej epoki (epoka 1000, seria 5800)

Oceniono, że większość tworzonych par liter naśladuje realne, ale zdarzają się niedoskonałe egzemplarze. Rzadkością są jednak bardzo źle wygenerowane, tak jak egzemplarz „fr” oznaczony na czerwono.

### 4.2.2.5 Podsumowanie

Oceniono, że wyniki zwracane po 1000 epokach są dobre. Model sieci pochodził z programu mającego służyć generowaniu cyfr. Poradził on sobie jednak na tyle dobrze, że został użyty jako sieć właściwa, a nie tylko punkt zaczepienia do dalszego rozwoju. Elastyczność ta pozawala przypuszczać, że program poradziłby sobie także z wszystkimi możliwymi parami.

Proces uczenia sieci przez 1000 epok zajmował łącznie około 240 godzin.

### 4.2.2.4 Wykresy obrazujące proces uczenia

W sieciach typu GAN nie mamy prostej możliwości na szacowanie postępu sieci. Dla sieci dyskryminacyjnych postęp nauki sprawdza się badając jak dobrze klasyfikowane są dane ze zbioru walidacyjnego (nie należące do danych treningowych). W przypadku sieci GAN chcemy oceniać jakość wygenerowanych danych i najczęściej robi to się poprzez wzrokowe ich sprawdzanie.

W jakimś stopniu postępy treningu można badać poprzez:

- Wypisywanie jak generator oceniał prawdziwe oraz wygenerowane dane.
- Wypisywanie wartości funkcji straty zarówno dla dyskryminatora jak i generatora.

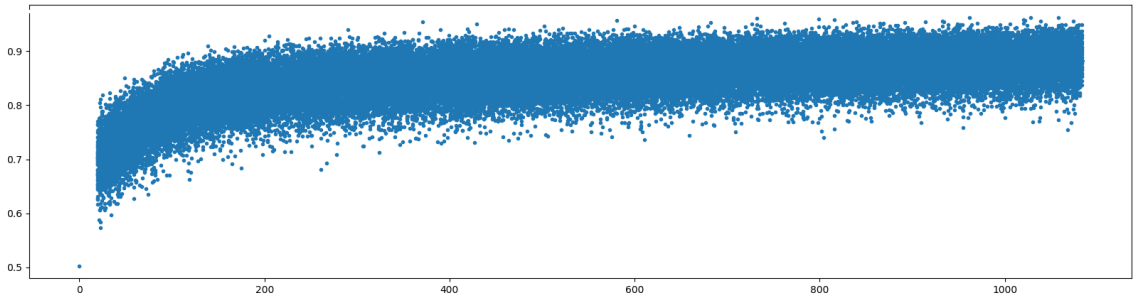
Informacje te nie muszą dobrze oddawać jakości wygenerowanych danych, gdyż umiejętność rozpoznawania dyskryminatora zmienia się z epoki na epokę. Obydwie sieci mogą być dobre i odnotowywać ciągły postęp, ale np. jedna z nich będzie o krok dalej niż druga i wypisywane wyniki będą słabe.

Teoretycznie w sieciach GAN generator uznaje prawdziwość danych ze zbioru oraz tych wygenerowanych na równym poziomie – po około 50%. Dzieje się tak ponieważ dyskryminator z jednej strony trenowany jest do rozpoznawania prawdziwych danych, z drugiej do odrzucania tych stworzonych. Jeżeli obydwa te zbiory są tak podobne, że aż nierozróżnialne, to można powiedzieć, że dyskryminator raz uczy się rozpoznawać prawdziwe dane, a potem je odrzucać i jego skuteczność zaczyna oscylować w okolicy 50%.

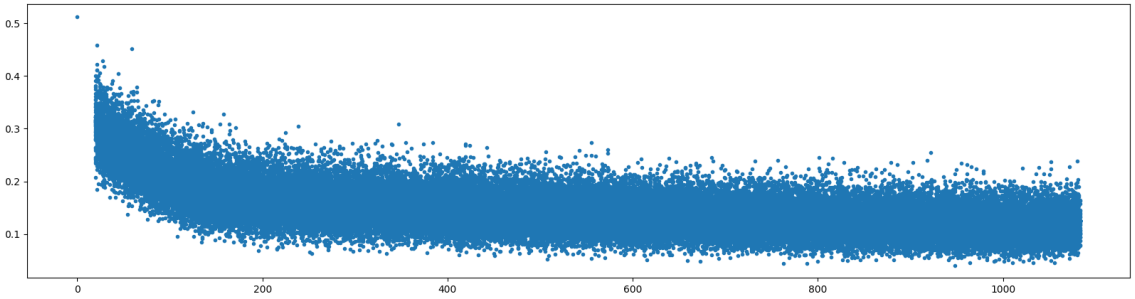
Praktycznie jednak często dzieje się tak, że skuteczność rozpoznania które dane są fałszywe, a które prawdziwe wynosi około 70-80% [43]. Wartości funkcji straty oscylują natomiast dla dyskryminatora 0.5-0.8, dla generatora 1.0-2.0 albo więcej [43]. Sytuacja taka może być spowodowana np. tym, że dyskryminator jest o krok dalej generatora oraz nauczył się rozpoznawać zbiór prawdziwych danych zbyt dokładnie (coś w rodzaju przeuczenia).

Poniższe wykresy obrazują proces uczenia pokazując odpowiednio:

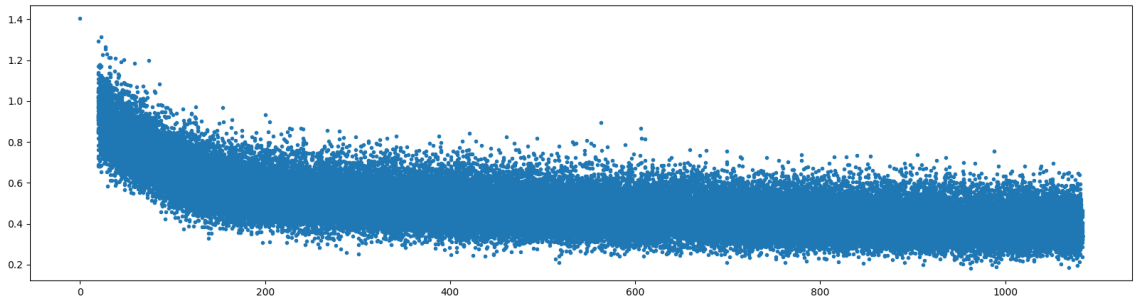
- Prawdziwość danych ze zbioru - Rysunek 29.
- Prawdziwość danych wygenerowanych - Rysunek 30.
- Wartość funkcji straty dla dyskryminatora - Rysunek 31.
- Wartość funkcji straty dla generatora - Rysunek 32.



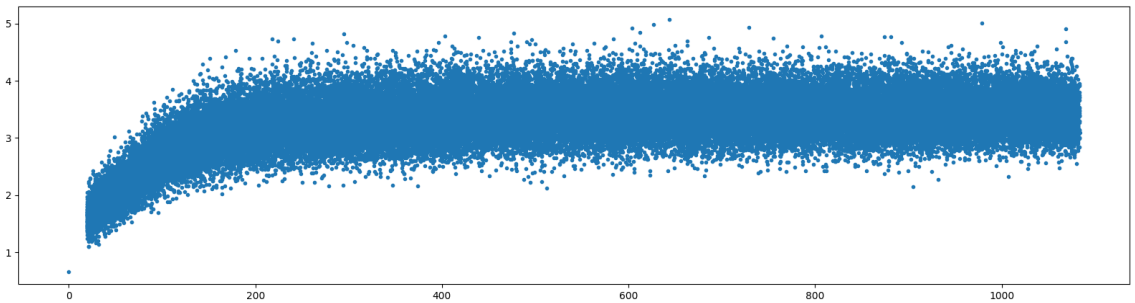

**Rysunek 29 Prawdziwość danych ze zbioru w zależności od numeru epoki.**




**Rysunek 30 Prawdziwość danych wygenerowanych w zależności od numeru epoki.**




**Rysunek 31 Wartości funkcji straty dla dyskryminatora w zależności od numeru epoki.**




**Rysunek 32 Wartość funkcji straty dla generatora w zależności od numeru epoki.**

Powyższe wykresy wygenerowane zostały przy pomocy zebranych danych i własnego programu używającego matplotlib.pyplot.

Rozpoznawanie które dane są prawdziwe, a które fałszywe oscyluje w okolicach powyżej 85% (dla danych wygenerowanych 0.15 oznacza 85% skuteczności w stwierdzeniu że są fałszywe). Funkcja straty dla dyskryminatora wynosi około 0.4, dla generatora trochę poniżej 3.5. Wartości te są praktycznie zgodne z tymi przytoczonymi kilka akapitów wyżej jako praktycznie spotykane. Mogą też wskazywać na prawdziwe problemy – dyskryminator wyłapuje sztuczność stworzonych danych. W celu poprawy tej statystyki można by było np. wydłużać proces uczenia lub poprawić architekturę sieci. Biorąc jednak pod uwagę rozdział: 4.2.2.3 Przegląd procesu uczenia, stwierdzono że nie obserwuje się dalszej poprawy w jakości wygenerowanych danych (nie ma potrzeby dalszego uczenia), a tworzone dane są wystarczająco dobre (nie jest konieczna zmiana architektury).

# Rozdział 5 Wyniki

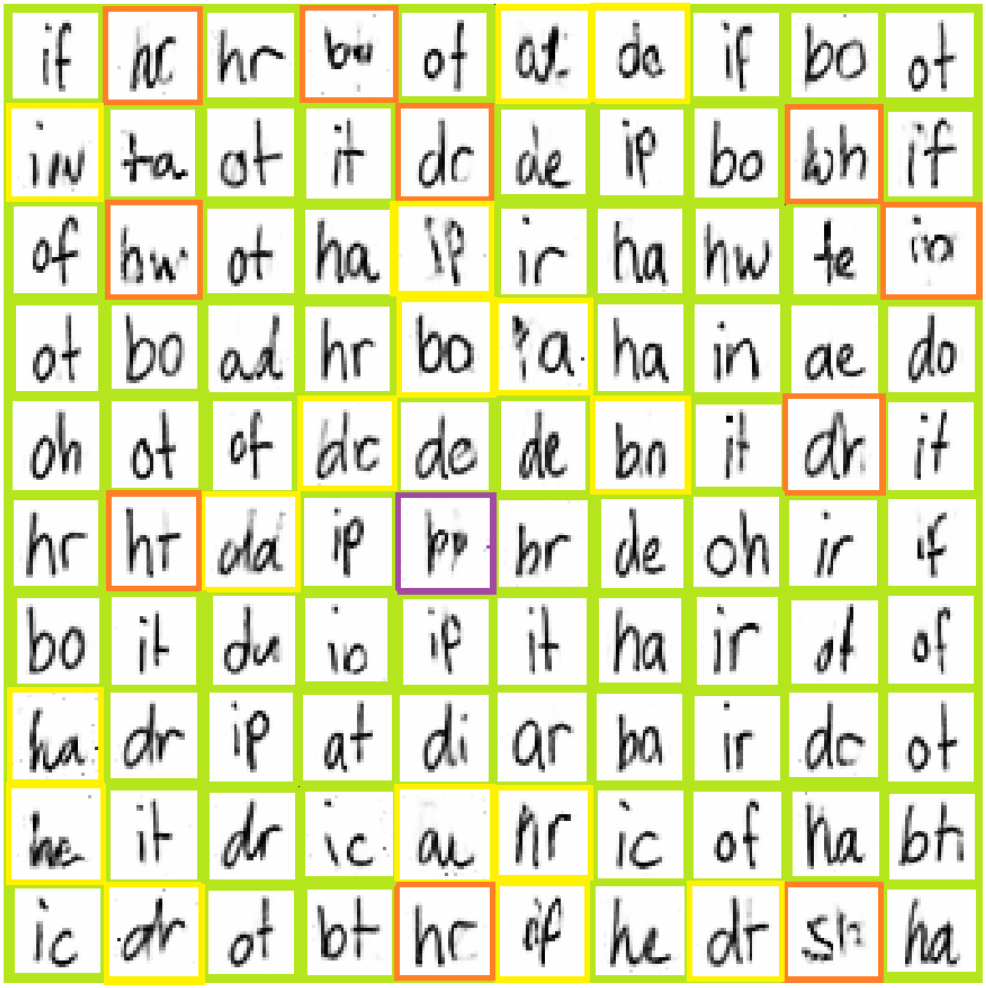
## *5.1 Ocena uzyskanych rezultatów*

Naukę sieci zakończono na 1083 epoce 1300 serii, gdy wzrokowo oceniono, że uczenie nie przynosi dalszych postępów.

Postanowiono sklasyfikować wyniki przypisując wygenerowanym egzemplarzom oceny zgodnie z następującymi zasadami.

- 0 – para nierozpoznana okiem.
- 1 – para rozpoznana, złej jakości
- 2 – para rozpoznana, nie rzucałaby się w oczy będąc zapisaną w prawdziwym tekście w małym rozmiarze.
- 3 – para rozpoznana, jakość zadowalająca w mniejszym rozmiarze.
- 4 – para rozpoznana, dobra jakość.

Oceny zobrazowano oznaczając poszczególne pary kolorami (Rysunek 33).



Rysunek 33: 100 wygenerowanych par liter wraz z ocenami (1- fioletowy, 2- pomarańczowy, 3-żółty, 4-zielony).

Należy zwrócić uwagę, że spore powiększenie ułatwia doszukiwanie się nieprawidłowości. System oceniania bierze to jednak pod uwagę.

Wyniki przedstawiają się następująco:

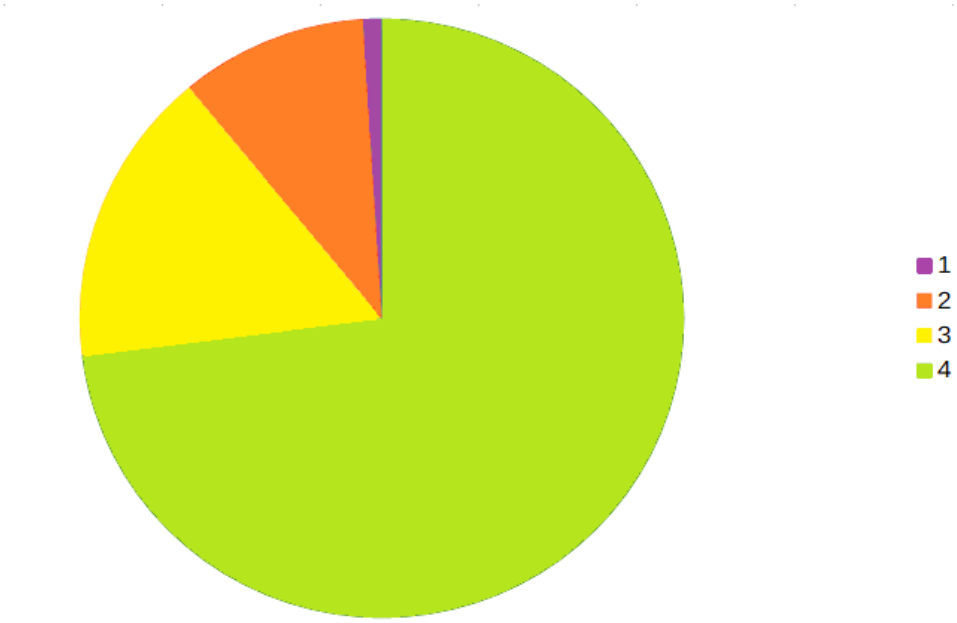
0 (czerwony) – brak, nie doszukano się pary niemożliwej do rozpoznania gołym okiem.

1 (fioletowy) – 1 (para uznana za „pp")

2 (pomarańczowy) – 10

3 (żółty) – 16

4 (zielony) – 73




Rysunek 34: Wykres kołowy przedstawiający ilość poszczególnych ocen.

Należy pamiętać o tym, że pomimo tego iż dane wejściowe tworzone były przez generator oraz dodatkowo sprawdzane przez inną sieć nieuronową, to obydwa te programy nauczyły się swoich zadań na podstawie ręcznego pisma (używano zbioru iam [29]). Takowe pismo nie jest bezbłędne. Założyć można, że komputerowe tworzenie nie do końca idealnego pisma jest naturalne i w jakimś stopniu pożądane. Zdaje się jednak że źle wygenerowane pary zdarzają się trochę zbyt często i jest to obszar w którym można by było dokonać usprawnień.

Oceniając obiektywnie łagodnie – należy stwierdzić, że tylko raz na kilkaset par zdarzają się nierozpoznawalne (ocena 0). W zbiorze 100 par pojawiła się zaledwie 1 para z oceną 1, która i tak mogłaby zostać niezauważona w normalnym tekście. Jakość aż 99 egzemplarzy jest do przyjęcia gdyby używać małych znaków, natomiast 73 z nich zostało ocenionych jako bardzo dobre.

# Podsumowanie i dalsze możliwości rozwoju

W pracy tej zmierzono się z problemem tworzenia pisma naśladującego odręczne za pomocą programu komputerowego, do czego należało wykorzystać metody sztucznej inteligencji. Rozwiązanie oparto na prężnie rozwijającej się gałęzi sieci neuronowych istniejącej dopiero kilka lat, czyli architekturze generatywnych sieci przeciwstawnych. Idealnym rozwiązaniem byłoby tworzenie całych wyrazów, pozwalające na generowanie całych zdań. Jednak wraz ze wzrostem ilości znaków ilość możliwych kombinacji rośnie wykładniczo, co mogłoby stanowić spory problem w rozwiązaniu wykorzystującym sieci typu GAN. Rozwiązania tworzące pojedyncze znaki były już dostępne i brakowało im jednej cechy ludzkiego pisma – ciągłości liter. Dlatego zdecydowano się na rozwiązanie pośrednie, czyli tworzenie par liter. Pozwala to zachować pożądaną cechę ciągłości oraz daje doświadczenie lub byłoby możliwe do wykorzystania w rozwiązania problemu generowania całych wyrazów.

Bardzo dużym problemem okazał się być brak zbioru danych zawierającego pary liter. Oceniono, że odpowiednie wycięcie par liter z prawdziwego tekstu będzie trudnym zadaniem. Zdecydowano się wykorzystać znaleziony generator oparty o inną sieć neuronową (architektura RNN). Nie było to bezproblemowe użycie i potrzebne były rozwiązania pomagające go wykorzystać. Przykładowo tworzenie i wycinanie ciągów znaków nadmiarowej wielkości, czy też użycie programów sprawdzających poprawność zbioru danych, m.in. kolejnej znalezionej sieci neuronowej przystosowanej do swojego zadania i wykorzystanej jako OCR. Problemy z generowaniem zbioru wymusiły ograniczenie się do 97 najlepiej tworzonych par liter.

Ostatecznie cel został osiągnięty a rezultaty pracy ocenia się jako dobre. Ocena byłaby bardzo dobra, jednak istnieje miejsce do poprawy statystycznej, czyli ilości bardzo dobrze wygenerowanych par liter.

Opisane poniżej pomysły na dalsze możliwości rozwoju, dotyczą poprawy tego generatora jak i wykorzystania jego efektów do generowania całych wyrazów.

1. Stworzenie zbioru danych zawierającego wszystkie możliwe pary liter (np. wycinanie i obróbka danych z prawdziwego tekstu) i wykorzystanie go w programie generującym również wszystkie możliwe pary.
2. Użycie innej architektury, np. głębokiej konwolucyjnej generatywnej sieci przeciwstawnej (ang. deep convolutional neural network). Architektura taka łączy potencjał sieci GAN wraz ze sporymi możliwościami sieci typu CNN i powinna przynieść poprawę w osiąganych rezultatach.
3. Wykorzystanie etykiet dokładnie opisujących poszczególne pary liter. Niektóre egzemplarze są źle wygenerowane dlatego, że są „pomiędzy” dwiema parami liter. Dla dyskryminatora nie są to bardzo złe egzemplarze, gdyż rozpoznaje on cechy należące do wszystkich par liter, a egzemplarze wygenerowane pomiędzy i tak będą je zawierać. Etykiety dokładnie opisujące jaka para liter będzie stworzona powinny znacząco ograniczyć ten problem.

4. Stworzenie programu wykorzystującego przedstawiony generator i potrafiącego łączyć pary liter w 3 literowe ciągi znaków, np. „ad” + „da” = „ada”. Zadanie to powinno być prostsze niż łączenie pojedynczych liter w pary, ze względu na istnienie łączeń pomiędzy parami liter oraz dodatkową literę za pomocą której można „skleić” dłuższy ciąg znaków. Następnie dalszy rozwój programu pozwalający na tworzenie wszystkich wyrazów.
5. Rozszerzenie możliwości użytej sieci GAN do generowania całych wyrazów.

# Bibliografia i referencje

Aktualność podanych linków sprawdzona na dzień 09.01.2020 r.


[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. *Generative Adversarial Networks* arXiv:1406.2661[stat.ML], https://arxiv.org/abs/1406.2661

[2] Alonso, Eloi & Moysset, Bastien & Messina, Ronaldo. (2019). *Adversarial Generation of Handwritten Text Images Conditioned on Sequences*, https://www.researchgate.net/publication/331485428_Adversarial_Generation_of_Handwritten_Text_Images_Conditioned_on_Sequences.

[3] Barnabas Poczos *Introduction to Machine Learning (Lecture Notes) Perceptron*, http://www.cs.cmu.edu/~10701/slides/Perceptron_Reading_Material.pdf

[4] Funkcja aktywacji, https://pl.wikipedia.org/wiki/Funkcja_aktywacji

[5] Prajit Ramachandran, Barret Zoph, Quoc V. Le, *Searching for Activation Functions*, arXiv:1710.05941 [cs.NE], https://arxiv.org/abs/1710.05941 https://arxiv.org/pdf/1710.05941.pdf

[6] Geoffrey Hinton with Nitish Srivastava, Kevin Swersky, *Overview of mini-batch gradient descent* http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov Dropout: *A Simple Way to Prevent Neural Networks from Overftting*. Toronto, 2014. http://jmlr.org/papers/v15/srivastava14a.html

[8] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, Sameep Tandon, *Optimization: Stochastic Gradient Descent*, http://deeplearning.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/

[9] Binarna entropia krzyżowa, torch.nn.BCELoss https://pytorch.org/docs/stable/nn.html?highlight=bceloss#torch.nn.BCELoss

[10] Brain-wiki contributors, *Uczenie maszynowe i sztuczne sieci neuronowe/Wykład 7*, https://brain.fuw.edu.pl/edu/index.php?title=Uczenie_maszynowe_i_sztuczne_sieci_neuronowe/Wyk%C5%82ad_7&oldid=6436.

[11] Jason Brownlee, *How to Develop a GAN for Generating MNIST Handwritten Digits*, https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/

[12] Branko Blagojevic, *Generating Letters Using Generative Adversarial Networks*, https://medium.com/ml-everything/generating-letters-using-generative-adversarial-networks-gans-161b0be3c229

[13] Yann LeCun, Corinna Cortes, Christopher J.C. Burges, THE MNIST DATABASE of handwritten digits http://yann.lecun.com/exdb/mnist/

[14] Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from http://arxiv.org/abs/1702.05373

[15] Alex Graves, *Generating Sequences With Recurrent Neural Networks*, https://arxiv.org/abs/1308.0850

[16] Grzegorz Opoka (Grzego), *handwriting-generation*, https://github.com/Grzego/handwriting-generation

[17] Matplotlib.pyplot, https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.html

[18] Pillow moduł ImageFilter
https://pillow.readthedocs.io/en/5.1.x/reference/ImageFilter.html
[19] Pillow, moduł Image, filtr Lanczos,
https://pillow.readthedocs.io/en/3.0.x/reference/Image.html?highlight=lanczos
[20] gazzawazza, *resizing algorithms*
https://newsgroup.xnview.com/viewtopic.php?t=20630
[21] helmut, *Reduce an image with text makes text blur*,
https://newsgroup.xnview.com/viewtopic.php?t=473
[22] Pillow, *Concepts*, filtr Lanczos,
https://pillow.readthedocs.io/en/3.0.x/handbook/concepts.html?highlight=lanczos
[23] Pillow, moduł Image, metoda convert
https://pillow.readthedocs.io/en/3.1.x/reference/Image.html
[24] PNG (ang. Portable Network Graphics)
https://pl.wikipedia.org/wiki/Portable_Network_Graphics
[25] PyTorch, zbiór MNIST, https://pytorch.org/docs/stable/torchvision/datasets.html
[26] PyTorch, DataLoader,
https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader
[27] Arlen Lu (Arlen0615), *Convert-own-data-to-MNIST-format*
https://github.com/Arlen0615/Convert-own-data-to-MNIST-format
[28] Myle Ott, (myleott), *mnist_png*
https://github.com/myleott/mnist_png/blob/master/convert_mnist_to_png.py
[29] U. Marti and H. Bunke. *The IAM-database: An English Sentence Database for Off-line Handwriting Recognition*. Int. Journal on Document Analysis and Recognition, Volume 5, pages 39 - 46, 2002., http://www.fki.inf.unibe.ch/databases/iam-on-line-handwriting-database/download-the-iam-on-line-handwriting-database
[30] Harald Scheidl (githubharald), *SimpleHTR*
https://github.com/githubharald/SimpleHTR
[31] Python multiprocessing, https://docs.python.org/3.6/library/multiprocessing.html
[32] Platforma PLGrid, http://plgrid.pl/
[33] Superkomputery dostępne na PLGrid
https://docs.cyfronet.pl/pages/viewpage.action?pageId=4260595
[34] Superkomputer Zeus
http://www.cyfronet.krakow.pl/komputery/13345,artykul,zeus.html
[35] TensorFlow, https://www.tensorflow.org/
[36] Diego Gomez Mosquera, *GANs from Scratch 1*, https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f
[37] Soumith Chintala, Emily Denton, Martin Arjovsky, Michael Mathieu, *ganhacks*,
https://github.com/soumith/ganhacks
[38] PyTorch, metoda randn, https://pytorch.org/docs/stable/torch.html#torch.randn
[39] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 [cs.LG], https://arxiv.org/abs/1412.6980
[40] PyTorch https://pytorch.org/
[41] Jupyter Notebook, https://jupyter.org/
[42] PyTorch, metoda save,
https://pytorch.org/docs/stable/torch.html?highlight=save#torch.save

[43] Jason Brownlee, *How to Identify and Diagnose GAN Failure Modes*, https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

[44] VisualParadigm, https://online.visual-paradigm.com/

[45] FooPlot, http://fooplot.com/

[46] Generator obrazu przedstawiającego zadaną sieć neuronową. http://alexlenail.me/NN-SVG/index.html

[47] Desmos, https://www.desmos.com/calculator