UMCS
UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **Informatyka**
Specjalność: **Sztuczna inteligencja**

Filip Ręka
nr albumu: 296595

Analiza porównawcza zastosowań tradycyjnych oraz wariacyjnych autoenkoderów

Comparative analysis of traditional and variational autoencoders' applications

Praca licencjacka
napisana w Katedrze Cyberbezpieczeństwa
pod kierunkiem dr hab. inż. Michała Wydry

Lublin 2021

Spis treści

Wstęp

Autoenkoder jest jednym z rodzajów sieci neuronowych, której zadaniem jest nauczenie się zakodowania nieoznaczonych danych. Kod jest wykorzystywany do ponownego, jak najlepszego, wygenerowania wejścia sieci. Autoenkoder uczy się reprezentacji zbioru danych jako zmiennych ukrytych przez ignorowanie nieistotnych części danych. Wariacyjne autoenkodery są popularnymi modelami generacyjnymi. Zostały zaproponowane przez Diederika P. Kingma i Maxa Wellinga w roku 2014 [1]. Najczęściej zostają one skategoryzowane do modeli uczenia częściowo nadzorowanego. Znajdują zastosowanie w generacji obrazów, tekstu, muzyki oraz w detekcji anomalii. W przeciwieństwie do tradycyjnych autoenkoderów prezentują probabilistyczne podejście do generowania zmiennych ukrytych. Swoją popularność zawdzięcza swojej budowie, która jest oparta na sieciach neuronowych oraz możliwości trenowania ich przy pomocy metod gradientowych.

Cel i zakres pracy

Celem pracy jest przegląd i ocena możliwości dwóch popularnych modeli uczenia maszynowego: tradycyjnego, oraz wariacyjnego autoenkodera. Oba modele, mimo podobieństwa w nazwie, różnią się budową oraz zastosowaniami. Jako modele kompresujące dane, są wykorzystywane do redukcji wymiarów, odszumiania danych oraz detekcji anomalii. Tradycyjna konstrukcja modelu, w przeciwieństwie do wariacyjnej, nie nadaje się do generacji danych takich jak obrazów lub ścieżek audio [2]. W pracy zostanie wytłumaczony ten problem i na podstawie matematycznych formuł wyprowadzona będzie struktura generacyjnego modelu. Dla obu modeli zostaną pokazane zastosowania, do których każdy model nadaje się najlepiej, oraz zostaną zaimplementowane w języku Python w bibliotece TensorFlow.

Rozdział 1

Tradycyjny autoenkoder

1.1 Informacje wstępne

Autoenkoder (AE) jest specyficzną wersją sieci neuronowej składającej się z dwóch części: enkodera, który koduje dane wejściowe oraz dekodera, który na podstawie kodu rekonstruuje wejście [3]. Architektura enkodera wymaga, aby jego warstwa wyjściowa generująca reprezentacje danych była mniejsza niż warstwa wejściowa. Często zwężenie to jest nazywane *bottleneck*. Model na swoją warstwę wejściową oraz wyjściową dostaje te same dane. Posiadając dane wejściowe X o wymiarze m oraz chcąc je zakodować do wymiaru n, można formalnie zapisać:

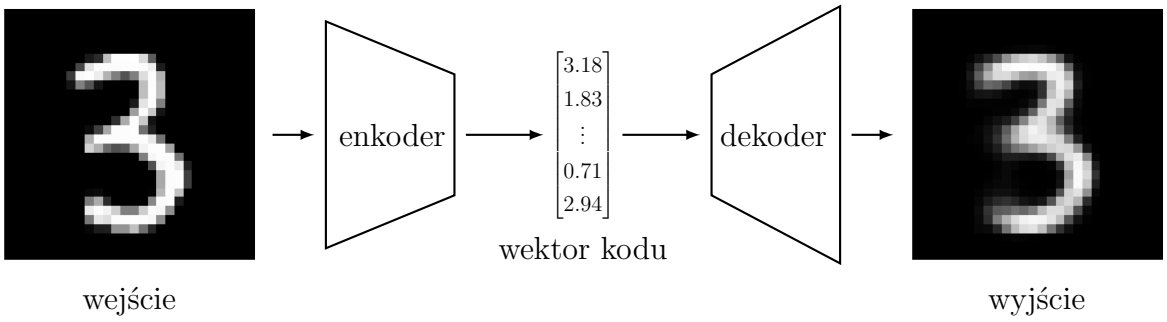
$$\text{Enkoder E: } \mathbb{R}^m \rightarrow \mathbb{R}^n$$
$$\text{Dekoder D: } \mathbb{R}^n \rightarrow \mathbb{R}^m$$
$$\text{gdzie } n < m$$

Celem *bottleneck-a* jest skompresowanie wejścia i zachowanie w ukrytych wartościach jak najwięcej informacji. W momencie, kiedy $n = m$, model przekazałby wartości z pierwszej warstwy na ostatnią bez potrzeby kompresji. Celem treningu całego autoenkodera jest zminimalizowanie błędu pomiędzy prawdziwymi danymi wejściowymi a tymi odkodowanymi ze skompresowanych wartości. W przypadku obrazów funkcją straty może być na przykład błąd średniokwadratowy zapisany w 1.1 lub binarna entropia krzyżowa, która powie, jak wynik różni się od wejścia.

$$\mathcal{L}(x, \hat{x}) = \frac{1}{m}\sum_{i=0}^{m}(x_i - \hat{x}_i)^2 = \frac{1}{m}\sum_{i=0}^{m}(x_i - D(E(x_i)))^2 \tag{1.1}$$

gdzie x jest wejściem, $\hat{x}$ jego rekonstrukcją, a m wymiarem danych

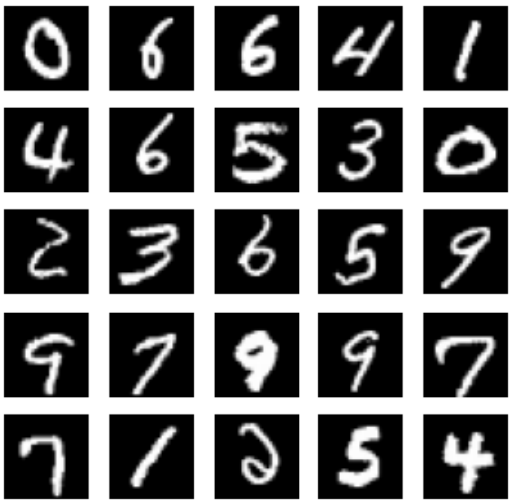
Rysunek 1.1 [3] pokazuje kompresje obrazu do wektora kodu, a następnie jego rekonstrukcje.



Rysunek 1.1: Schemat budowy autoenkodera

1.1.1 Zbiór danych MNIST

Zbiór danych Modified National Institute of Standards and Technology (MNIST) jest zbiorem wielu odręcznie pisanych cyfr [4]. Znajduje szerokie zastosowanie w nauce i prezentacjach możliwości modeli uczenia maszynowego. W jego skład wchodzi 60 000 obrazów przeznaczonych do treningu modeli oraz 10 000 do testów. Obrazy są czarno-białe i mają wymiary 28 na 28 pikseli.

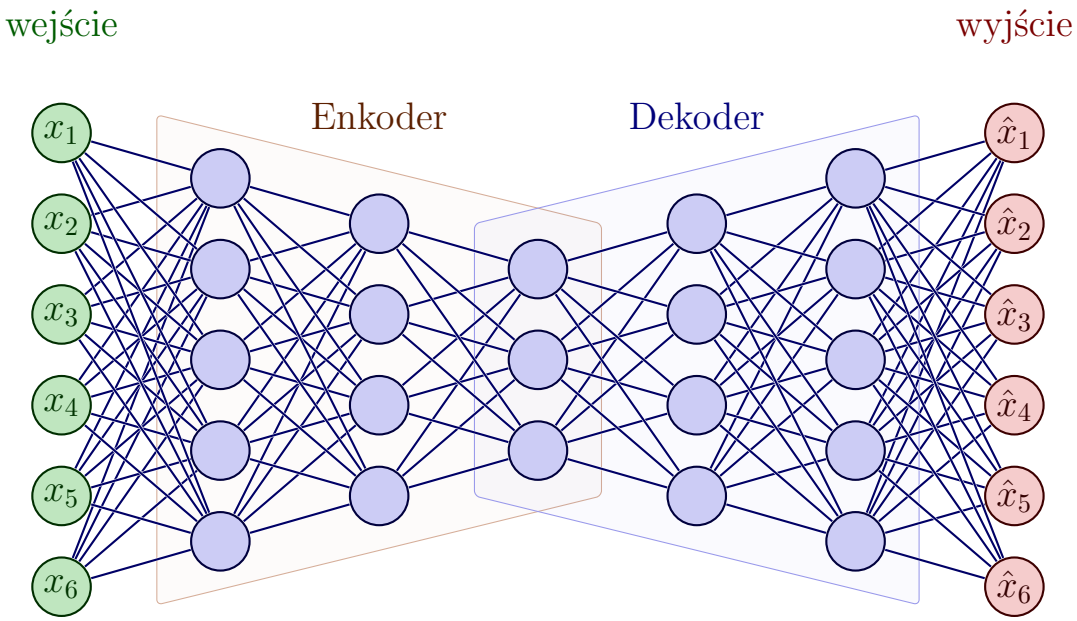


Rysunek 1.2: Przykładowe obrazy ze zbioru danych

W pracy zbiór ten jest wykorzystywany w celu pokazania różnic w budowie i zastosowaniu tradycyjnych oraz wariacyjnych autoenkoderów, dlatego, że na obrazach łatwo zobaczyć zachowanie kompresji i rekonstrukcji. Cyfry są również bardzo łatwe do sklasyfikowania dla człowieka, przez co nowe, wygenerowane dane są proste do porównania z tymi, należącymi do zbioru danych.

1.2 Budowa

W skład autoenkodera wchodzi enkoder oraz dekoder. Obie części są w pełni połączonymi sieciami neuronowymi, połączonymi również pomiędzy sobą. Prosta budowa sprawia, że bez problemu obie sieci są trenowane równocześnie.



Rysunek 1.3: Wizualizacja sieci tworzącej autoenkoder

Obrazek 1.3 [5] przedstawia prosty autoenkoder kompresujący sześciowymiarowe wejście do kodu o długości trzy. Enkoder, jak i dekoder mają dwie warstwy ukryte. Odbicie lustrzane architektury nie jest konieczne, aby model działał poprawnie, jednak zwyczajem jest używanie takiej architektury.
Hiperparametramy modelu, które można ustalić przed jego treningiem to:

- Ilość warstw ukrytych - jeśli że nasze dane są skomplikowane, dodatkowe warstwy ukryte będą miały pozytywny wpływ na otrzymywane rezultaty, ponieważ większa ilość warstw sprawia, że model jest w stanie nauczyć się bardziej skomplikowanych funkcji [6, 7].

- Ilość neuronów w poszczególnych warstwach - autoenkoder powinien posiadać w każdej warstwie mniej neuronów niż w poprzedniej. W ten sposób model nie będzie "oszukiwał" i zostaje zmuszony do reprezentacji jak najlepszej kompresji.

- Funkcja straty - najlepszymi funkcjami straty do treningu autoenkodera jest błąd średniokwadratowy lub binarna entropia krzyżowa w przypadku, kiedy dane są w przedziale od 0 do 1.

- Rozmiar kodu - jest to najistotniejszy parametr wybierany przed treningiem. Dłuższy kod oznacza zachowanie więcej istotnych elementów, a co za tym idzie lepsze

odwzorowanie przez dekoder. Z drugiej strony, używając dłuższego wektora zmiennych ukrytych, dostaje się gorszą kompresje danych. Długość kodu trzeba dobrać w zależności od problemu, który będzie rozwiązywany przy pomocy modelu.

1.3 Zastosowania

1.3.1 Redukcja wymiaru

Ilość otaczającej nas informacji, sprawia, że można z niej wyciągać coraz więcej ilości danych. Wiele z tych danych, takie jak obrazy, tekst czy nagrania są opisywane przez wiele parametrów. Redukcja wymiaru jest procesem zmniejszenia liczby zmiennych przeznaczonych do analizy, a zarazem zachowanie w nich jak najwięcej istotnych informacji. Powody, dla których proces zmniejszenia ilości wymiarów danych jest bardzo często wykorzystywane to między innymi:

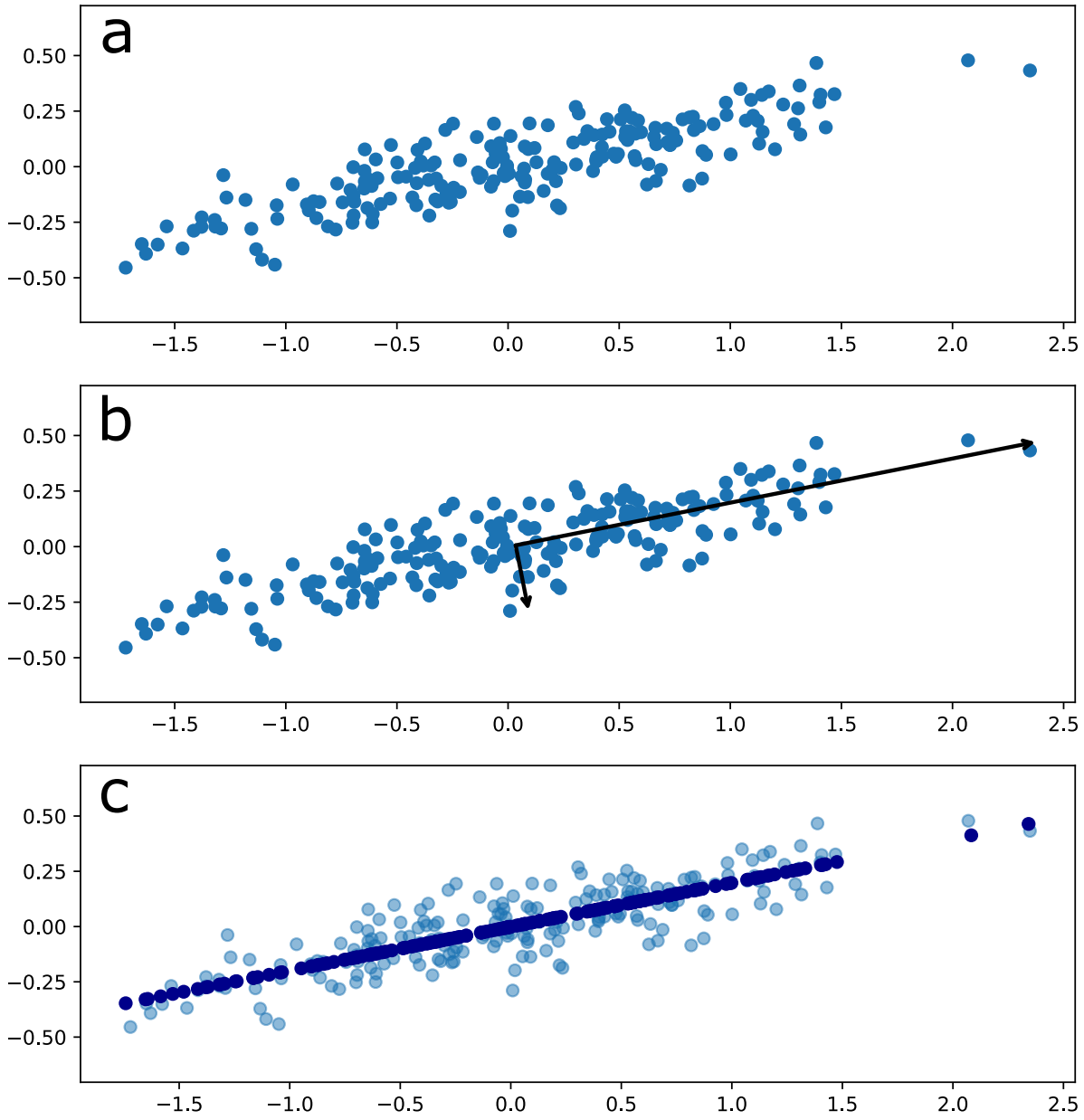
- Część zmiennych opisująca dane jest ze sobą nadmiernie skorelowana lub niesie ze sobą cechy, które nie są istotne statystycznie i usunięcie ich nie wpływa na poprawę działania modeli oraz czas ich treningu.
- Dane bardzo wysokiego wymiaru są trudne do analizy lub operacje na nich zajmują tak dużo czasu i zasobów, co sprawia, że stają się one bezużyteczne.
- Wielowymiarowe dane jest ciężej zwizualizować. Można je zredukować do jedno-, dwu- lub trzywymiarowej reprezentacji, co pozwoli na proste narysowanie wykresu, który będzie prosty do zrozumienia.

Autoenkoder nie jest jednym sposobem na redukcję wymiarów. Najbardziej rozpowszechnioną metodą jest *PCA*.

Analiza składowych głównych

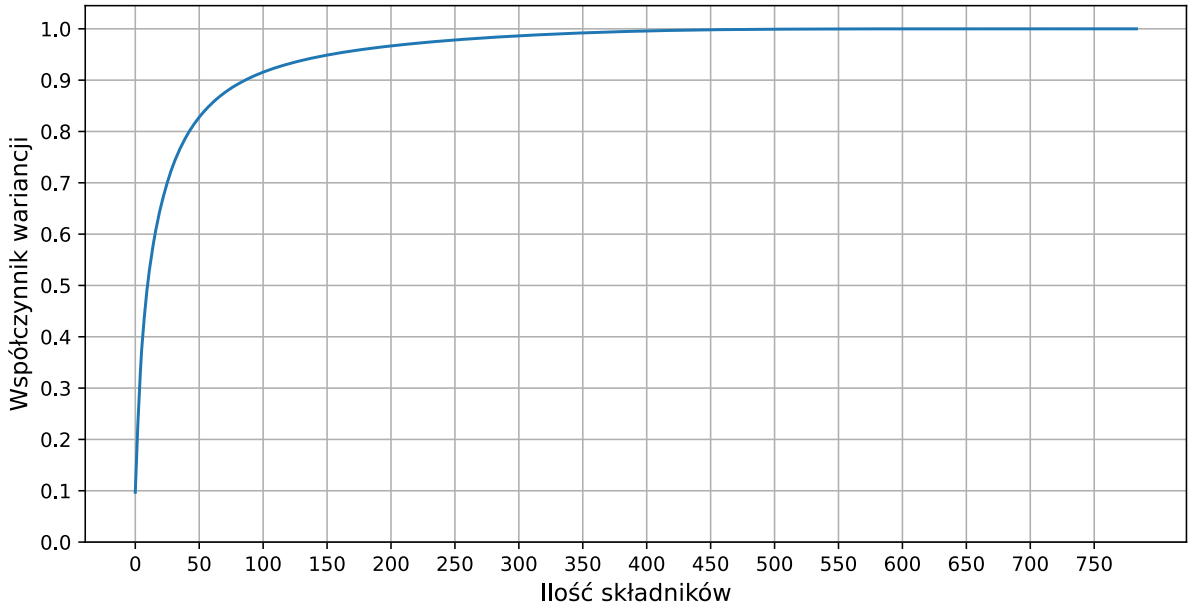
Analiza składowych głównych (*ang. Principal Components Analysis,* ***PCA***) służy do wyznaczania jak najmniejszej ilości nowych zmiennych, mówiących jak najwięcej o zbiorze danych. Wielowymiarowe dane koncentrują się w pewnych podprzestrzeniach oryginalniej przestrzeni. Analiza PCA pozwala znaleźć te podprzestrzenie, które są wektorami pełniącymi rolę nowych osi, które lepiej opisują nasz zbiór danych [8]. Omawiana metoda ogranicza się jedynie do przekształceń liniowych. Ilość wektorów, względem których można zredukować dane, jest równa wymiarowi danych. Obrazek b na rysunku 1.4 przedstawia właśnie te wektory na przykładzie dwuwymiarowego zbioru punków. Linia, względem której spłaszczane są dane, jest tą, która minimalizuje odległość do niej od wszystkich punktów. Wykres c na rysunku 1.4 pokazuje już zredukowane dane do jednego wymiaru. Nowe wektory są wybierane w taki sposób, aby wariancja rzutów poszczególnych

obserwacji była jak największa, co gwarantuje nam odwzorowanie jak największej ilości danych. Każdy kolejny wektor zachowuję się w taki sam sposób oraz jest ortonormalny do poprzednich.



Rysunek 1.4: Wizualizacja PCA na zbiorze punktów w przestrzeni dwuwymiarowej

Wykres 1.5 pokazuje zależność między ilością składników PCA a procentem wariancji opisywanym przez składniki na podstawie zbioru danych MNIST. Ilość składników mieści się w przedziale od 1 do 784 (28 razy 28 pixeli). Jak można zauważyć, dane reprezentowane przez około 80 wartości są w stanie opisać 90% wariancji danych, co jest znaczą redukcją z 784. Przy 400 składnikach osiągnięto praktycznie 100% pokrycia.



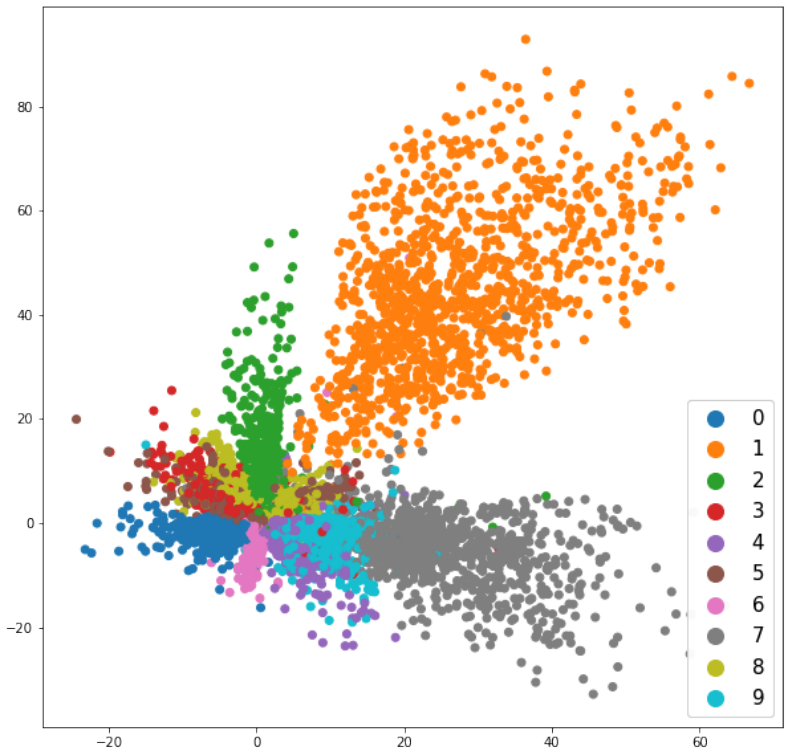
Rysunek 1.5: Procent wariancji opisywany przez ilość składników

Autoenkoder

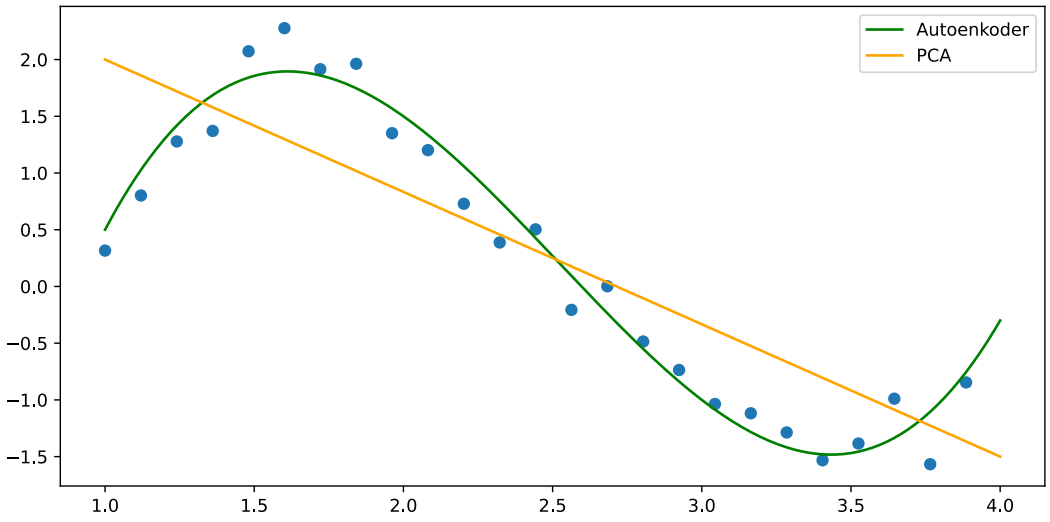
Użycie autoenkodera jest jedną z możliwości dokonania redukcji wymiarów. Jego budowa wymusza naukę jak najlepszej reprezentacji danych w kodzie. Zadaniem enkodera jest zachowanie w zmiennych jak najwięcej informacji dotyczących wejścia, a dekoder odwzorować jak najwięcej z nich. Najważniejszą cechą autoenkodera jest możliwość nauki przekształceń zarówno liniowych, jak i nieliniowych w zależności od doboru funkcji aktywacji. Różnicę pomiędzy oboma przekształceniami pokazano na rysunku 1.7 [10].

Porównanie obu metod

Obie przedstawione metody redukowania wymiarów są bardzo dobrymi wyborami. Wybór jednej z nich powinien zostać dopasowany do potrzeb i do zbioru danych, który jest używamy. PCA, będąc ograniczone do jedynie liniowych przekształceń, jest szybsze niż autoenkoder, który wymaga treningu [9]. Jest ono też lepszym wyborem, kiedy nasze dane są nadmiernie skorelowane. Rozpatrując przykład zbioru danych opisującego takie cechy ludzi, jak wzrost, waga, kolor oczu oraz długość włosów, oczywiste jest, że czyjaś waga jest zależna od wzrostu. PCA w takim przykładzie bez problemu odnajdzie tą zależność i zredukuje. Autoenkoder jest lepszym wyborem, kiedy dane są o wiele bardziej złożone, czyli obrazy lub pliki audio. Jednowarstwowy autoenkoder z liniową funkcją aktywacji na każdej warstwie zachowuje się dokładnie tak samo jak PCA [11].



Rysunek 1.6: Przestrzeń dwuwymiarowej zmiennej ukrytej dla zbioru MNIST

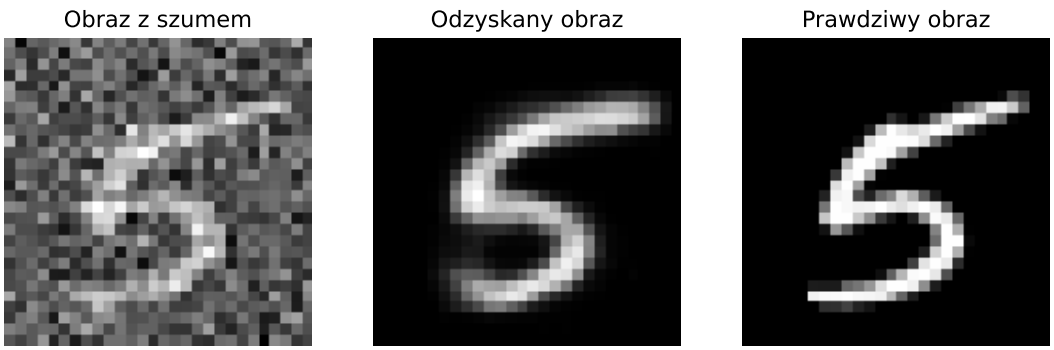


Rysunek 1.7: Liniowe oraz nieliniowe przekształcenie

1.3.2 Odszumianie obrazów

Model autoenkodera przeznaczony do odszumiania danych często dostaje swoją nazwę i jest określany mianem *Denoising autoencoder* (DAE). Tak jak zwykły autoenkoder, próbuje w jak najlepszy sposób skompresować dane, zachowując jak najwięcej istotnych informacji. Najważniejszą różnicą między tymi modelami są dane, które dostają na wejście i wyjście. Obrazy przyjmowane na warstwę wejściową są zaszumione, natomiast te z war-

stwy wyjściowej pochodzą prosto ze zbioru danych. Powodem, dla którego autoenkodery tak dobrze nadają się do odszumiania, jest ich umiejętność kompresji danych. Kompresja, której dokonują te modele jest stratna. W przypadku, kiedy głównym zadaniem jest jak najlepsze odtworzenie danych wejściowych, jest to kłopot, jednak w tym przypadku można wykorzystać tę własność. Porównując wyjście modelu z danymi bez szumu, zapewniamy, że model nauczy się w jakimś stopniu odtwarzać je poprawnie. Zmusza to w ten sposób model do ignorowania nieistotnych części danych oraz zapamiętywanie tylko tych, na podstawie których będzie możliwe jak najlepsze odwzorowanie wejścia sieci. Rysunek 1.8 pokazuje możliwości modelu na przykładzie zbioru danych MNIST. Do obrazów przeznaczonych na trening został dodany szum. Zaszumiony obraz powstał przez dodanie do oryginalnego obrazu losowo wybranych wartości z rozkładu Gaussa $\mathcal{N}(0, 1)$ przemnożonych przez stałą, która w tym przypadku wynosi $0,4$. Następnie obrazy wejściowe zostały skompresowane do kodu o długości 5, a następnie odkodowane przez dekoder.



Rysunek 1.8: Obraz z szumem, odzyskany oraz prawdziwy

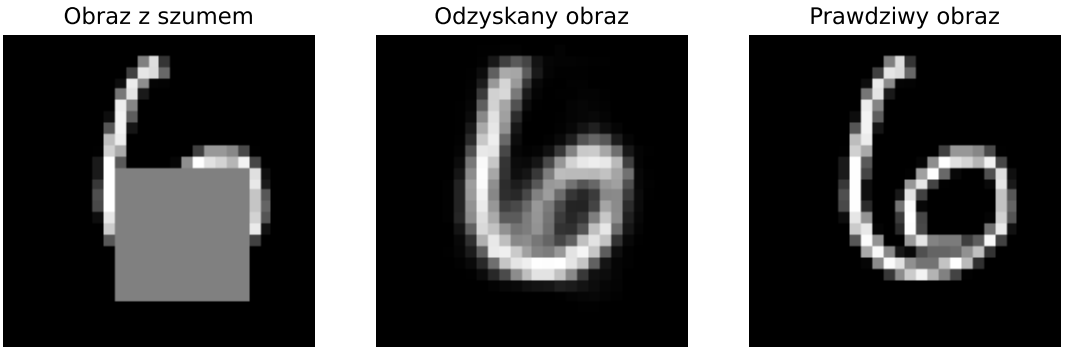
Wadą autoenkodera przeznaczonego do odszumiania danych jest jego ścisłe powiązanie ze zbiorem danych, na których został wytrenowany. Model z parametrami wytrenowanymi na jednym zbiorze danych nie będzie nadawał się do innego zbioru, z którego danymi będzie pracował. Jedynym rozwiązaniem tego problemu jest stworzenie nowego modelu przeznaczonego do użytku na nowych danych.

1.3.3 Uzupełnianie obrazów

Uzupełnianie obrazów ma na celu wypełnienie brakującej lub zamaskowanej części obszaru. Człowiek jest w stanie sobie poradzić z tym zadaniem bez problemu, jednak dla komputera nie jest ono oczywiste. Bierze się to z tego, że jest ogromna ilość możliwości wypełnienia nawet niewielkiej brakującej przestrzeni. Można wyróżnić dwa główne podejścia wypełniania obrazów:

- sieć posiada informację, w którym miejscu obrazu jest luka;
- sieć musi sama się nauczyć, które miejsce obrazu musi wypełnić.

Rysunek 1.9 przedstawia drugie podejście.



Rysunek 1.9: Obraz z luką, odzyskany oraz prawdziwy

1.4 Problemy z generacją nowych danych

Dobrym pytaniem jest, czy przy pomocy kodu można generować nowe dane podobne do tych, na których model został wytrenowany. Wiadomo, że sieć po treningu, jest w stanie ze zmiennych ukrytych odkodować obraz, więc ustawiając wejście dekodera na losowy punkt z przestrzeni zmiennych, powinno się dostać obraz, który jest podobny do tych, na których sieć została wytrenowana. Aby model mógł generować nowe dane, muszą zostać spełnione dwa warunki:

- Nasza przestrzeń kodu (tzw. zmiennych ukrytych) musi być ciągła, co znaczy, że dwa punkty znajdujące się obok siebie będą dawać podobne dane kiedy zostaną odkodowane.
- Przestrzeń musi być kompletna, co znaczy, że punkty wzięte z dystrybucji muszą dawać wyniki mające sens.

Tradycyjna architektura nie zapewnia przed treningiem żadnego z tych warunków. W dodatku nie znamy jaka jest dystrybucja według, której enkoder wybiera zmienne. Spoglądając na rysunek 1.6, widzimy, że przestrzeń zawiera luki. Szczególnie dobrze to widać między klasami oznaczającymi jedynki oraz siódemki. Kolejnym problemem widocznym na tej grafice jest brak separacji między klasami. Niektóre z nich są dobrze odseparowane od siebie, jednak inne całkowicie na siebie nachodzą, jak siódemki z dziewiątkami czy trójki z piątkami. Zadaniem modelu jest jak najlepsze odzwierciedlenie skompresowanych danych, a nie dbanie o to, czy rozkład zmiennych kodu spełnia przedstawione warunki. Może się tak zdarzyć, że sieć nauczy się akurat takiej dystrybucji, która pasuje, ale jest to bardzo mało prawdopodobne. Chcąc zbudować model generacyjny, trzeba mieć zagwarantowane, że za każdym razem rozkład będzie spełniał odpowiednie warunki.

1.5 Wnioski

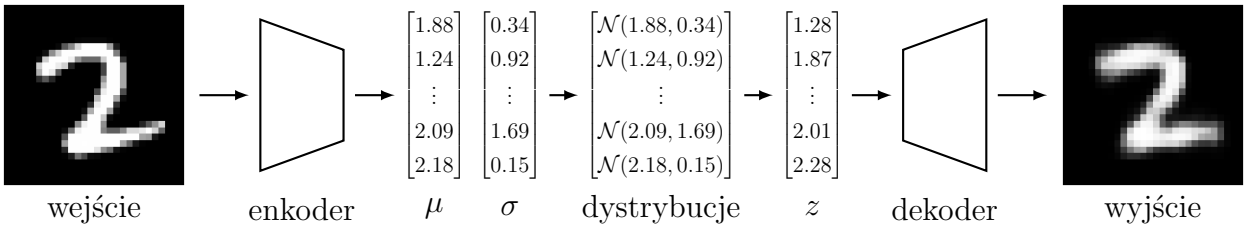
Autoenkoder, jako zwyczajna sieć neuronowa o zwężonej budowie, najlepiej sprawdza się w problemach, w których zależy nam na kompresji i rekonstrukcji danych. Kompresja wejścia modelu do kodu, pozwala na zachowanie tylko najistotniejszych elementów, na podstawie których, będzie można zrekonstruować je jak najlepiej. Zadania opierające się na tym procesie to między innymi: redukcja wymiarów, usuwanie szumu czy detekcji anomalii. Każdy ten przykład, opiera się na tej samej zasadzie działania polegającej na kompresji. Model porównując ze sobą dane wejściowe oraz te, które pochodzą z warstwy wyjściowej, jest w stanie po wielu epokach treningu zacząć generalizować, co pozwala na ignorowanie elementów, które są nieistotne w odtwarzaniu danych wejściowych. Enkoder tradycyjnego autoenkodera, nie kontroluje według jakiej dystrybucji rozłożona jest przestrzeń kodu. Chcąc wygenerować nowe dane, nie wiadomo, które jej punkty będą w stanie wygenerować podobne dane, ponieważ przestrzeń kodu może posiadać luki.

Rozdział 2

Wariacyjny autoenkoder

2.1 Informacje ogóle

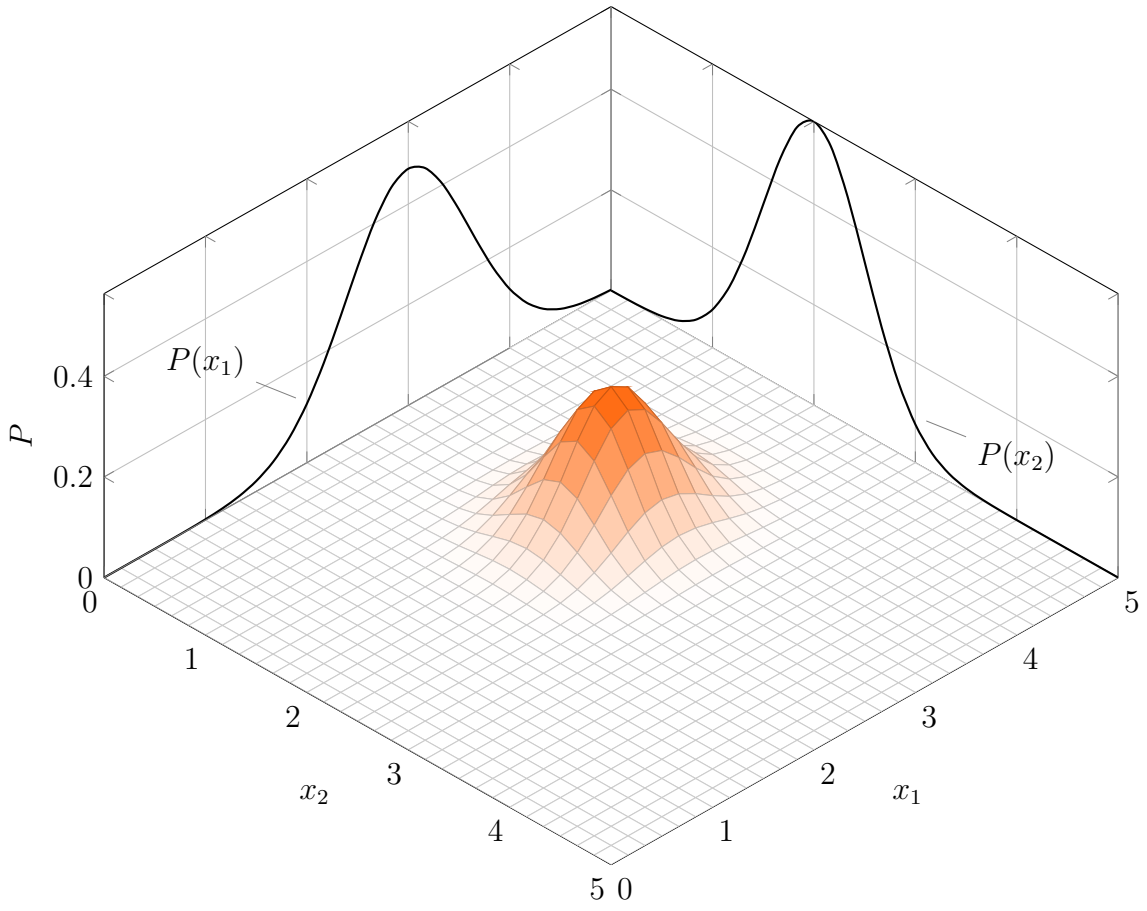
Wariacyjny autoenkoder (VAE) rozwiązuje problemy generacyjne tradycyjnego modelu. VAE ma na celu skompresowanie danych do określonego wielowymiarowego rozkładu ukrytego, a następnie z próbki tej dystrybucji próbuje jak najlepiej zrekonstruować wejście. Model ten należy go grupy wariacyjnych metod Bayesowskich, czemu zawdzięcza swoją nazwę. Dystrybucjami najczęściej wybieranymi do reprezentacji zmiennych ukrytych są rozkłady normalne. Rozkład normalny jest opisywany przy pomocy dwóch wartości: średnia, która oznaczana jest znakiem μ oraz odchylenie standardowe oznaczane σ. Jeśli dane zostaną skompresowane do kodu o długości n, enkoder wygeneruje dwa wektory n-wymiarowe, z którego jeden będzie przechowywał wartości średniej, a drugi odchylenia standardowego dla każdego z n rozkładów normalnych co przedstawia rysunek 2.1 [12].



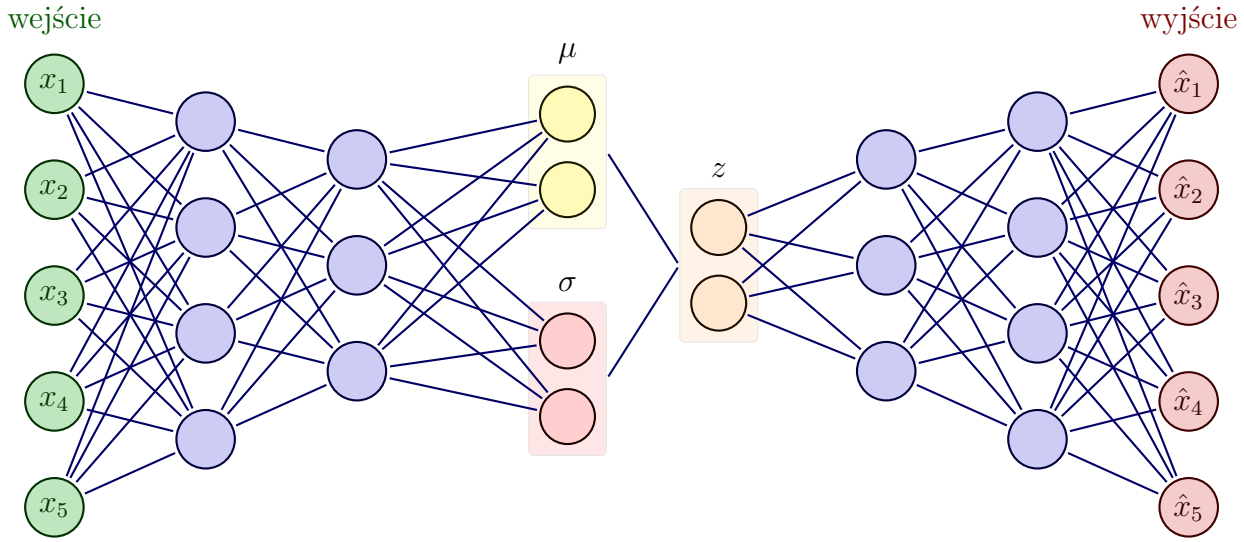
Rysunek 2.1: Schemat budowy wariacyjnego autoenkodera

Ograniczając enkoder do nauki wyłącznie tej dystrybucji, z której losuje zmienne ukryte, na podstawie których rekonstruuje on obrazy, zapewnia się gładką oraz ciągłą dystrybucje zmiennych losowych. Dekoder patrząc na obraz wyjściowy i porównując go z prawdziwym, nauczy się odkodowywać niedaleko oddalone od siebie punkty z rozkładu w bardzo podobny sposób.

Sieć neuronowa budująca wariacyjny autoenkoder posiada dość nietypową budowę. Ostatnia ukryta warstwa enkodera łączy się z dwiema niepołączonymi ze sobą warstwa-



Rysunek 2.2: Wielowymiarowy rozkład zmiennej ukrytej



Rysunek 2.3: Sieć neuronowa budująca model VAE

mi reprezentującymi średnią oraz odchylenie standardowe normalnej dystrybucji, której enkoder chce się nauczyć. Obie warstwy łączą się w jedną, która dokonuje operacji próbkowania z dystrybucji na nauczonych parametrach. Warstwy reprezentujące średnią, odchylenie standardowe oraz zmienne ukryte muszą posiadać taką samą liczbę neuronów,

przedstawiającą długość wektora reprezentującego skompresowane dane.

2.2 Matematyczny opis modelu

Model VAE mimo swojego podobieństwa do tradycyjnego autoenkodera znacznie różni się w opisie matematycznym. Największe różnice są obserwowane w zachowaniu enkodera. Kod, który on produkuje, nazywany jest wartościami ukrytymi, ponieważ są one wnioskowane na podstawie każdej danej. Trenując tradycyjny model, nie interesuje nas dystrybucja $p(z|x)$, z której pochodzą zmienne. Enkoder wariacyjnego autoenkodera jest, jak już zostało powiedziane, zobowiązany do nauki dystrybucji naszego wyboru. Wyrażenie $p(z|x)$ jest rozumiane jako dystrybucja generująca zmienne ukryte na podstawie przedstawionej danej x. Trzeba policzyć ten rozkład. Twierdzenie Bayesa mówi, że:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{2.1}$$

Aby obliczyć rozkład marginalny $p(x)$, trzeba policzyć:

$$p(x) = \int_z p(x|z)p(z)dz \tag{2.2}$$

Obliczenie tej całki jest bardzo trudne lub nawet obliczeniowo niemożliwe w rozsądnym czasie, ponieważ z jest często wielowymiarowym wektorem, a trzeba całkować po wszystkich wymiarach. Aby próbować policzyć tę całkę w inny sposób, można wybrać jedną z dwóch metod:

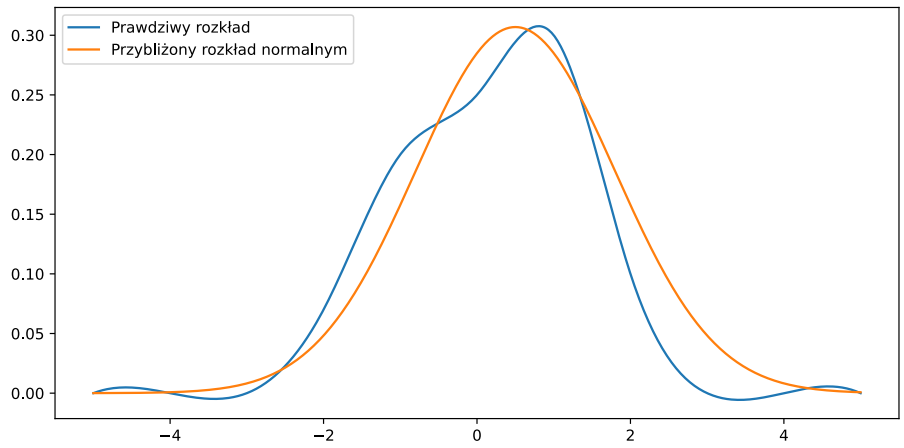
- próbkowanie Monte Carlo łańcuchami Markowa;
- wnioskowanie wariacyjne.

Przestrzeń przeszukiwań może być kombinatorycznie za duża, aby korzystać z pierwszej metody lub błąd przybliżenia tej całki będzie za duży w przypadku znacznej ilości wymiarów [13].

2.3 Wnioskowanie wariacyjne

Wnioskowanie wariacyjne pozwala zastąpić jedną dystrybucje, o której nie wiadomo za dużo i trudno z nią pracować na taką, jaka pasuje do problemu oraz dobrze odzwierciedla początkową [14]. Używając tej metody, uda się rozwiązać problem policzenia rozkładu $p(z|x)$. Zastąpiony będzie rozkładem q, który jak najlepiej odwzoruje oryginalny rozkład.

Wykres 2.4 przedstawia dwie krzywe. Pomarańczowa krzywa, opisująca rozkład normalny w bardzo dobry sposób, przybliża prawdziwą dystrybucję.



Rysunek 2.4: Prawdziwa oraz przybliżona dystrybucja

2.3.1 Dywergencja Kullbacka-Leiblera

Aby być w stanie zminimalizować błąd pomiędzy prawdziwą a zamienną dystrybucją, należy posiadać miarę, która określa rozbieżność między dwoma rozkładami prawdopodobieństwa. Miarą tą jest dywergencja Kullbacka-Leiblera (D_{KL}). Podstawowymi jej własnościami są:

- jej wartość jest nieujemna; $D_{KL}(P\|Q)) \geqslant 0$. Miara przyjmuje wartość 0 tylko w przypadku kiedy P i Q są identycznymi dystrybucjami.
- miary tej nie można określić mianem metryki, ponieważ nie jest symetryczna ($D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$);

Celem jest zminimalizowanie błędu, co sprawi, że rozkład $q(z|x)$ będzie jak najbardziej podobny do $p(z|x)$.

$$q^*(z|x) = \underset{q(z|x)\in Q}{\operatorname{argmin}}(D_{KL}(q(z|x)\|p(z|x))) \tag{2.3}$$

gdzie Q to rodzina prostych dystrybucji, na przykład rozkładów Gaussa.

2.3.2 Dolna granica dowodów

Wzór na dywergencję jest zapisywany jako:

$$D_{KL}(q(z|x)\|p(z|x)) = \int_z q(z|x) \log \frac{q(z|x)}{p(z|x)} dz \tag{2.4}$$

Policzenie $q(z|x)$ jest proste, ponieważ sami dobieramy, jaką dystrybucją jest q. Dal modelu VAE najczęściej jest to rozkład normalny. Mimo to, obliczenia nie są możliwe, ponieważ znowu występuje problem wyznaczenia $p(z|x)$. Tym razem można przepisać:

$$p(z|x) = \frac{p(x,z)}{p(x)} \tag{2.5}$$

Podstawiając zapisaną inaczej wartość p do wzoru na dywergencję otrzymuje się [15]:

$$D_{KL}(q(z|x)\|p(z|x)) = \int_z q(z|x) \log \frac{q(z|x)}{p(x,z)} p(x) dz \tag{2.6}$$

Własności logarytmów pozwala zamienić iloczyn pod logarytmem na sumę w następujący sposób:

$$D_{KL}(q(z|x)\|p(z|x)) = \int_z q(z|x) \left(\log \frac{q(z|x)}{p(x,z)} + \log p(x) \right) dz \tag{2.7}$$

Mnożąc $q(z|x)$ przez nawias, otrzymaną sumę pod całką, można zapisać jako sumę dwóch całek. Następnie wartość $\log p(x)$ jest wyłączana przed całkę, ponieważ całkując po z, jest on traktowany jako stała.

$$D_{KL}(q(z|x)\|p(z|x)) = \int_z q(z|x) \left(\log \frac{q(z|x)}{p(x,z)} \right) dz + \log p(x) \underbrace{\int_z q(z|x) dz}_{\alpha} \tag{2.8}$$

Wartość całki zaznaczonej jako α jest równa 1, dlatego, że q jest funkcją dystrybucji prawdopodobieństwa, a warunek $z|x$ nie ma znaczenia, ponieważ całka jest liczona po z.

$$D_{KL}(q(z|x)\|p(z|x)) = \underbrace{\int_z q(z|x) \left(\log \frac{q(z|x)}{p(x,z)} \right) dz}_{\mathcal{L}} + \log p(x) \tag{2.9}$$

Na razie nie dokonano żadnych obliczeń, jedynie przepisano wzór na dywergencję.

$$\begin{aligned} D_{KL}(q(z|x)\|p(z|x)) &= \mathcal{L} + \log p(x) \\ -\mathcal{L} &= \log p(x) - D_{KL}(q(z|x)\|p(z|x)) \end{aligned} \tag{2.10}$$

Wartość $\log p(x)$ nosi nazwę dowodu, ponieważ mówi o prawdopodobieństwie otrzymania obserwacji x przez nasz model. Parametr $-\mathcal{L}$ jest nazywany dolną granicą dowodu *Evidence Lower BOund,* ***ELBO***. Nazwa pochodzi od własności, mówiącej o tym, że jej wartość jest zawsze mniejsza lub równa dowodowi. Własność ta bierze się z faktu, że dywergencja jest zawsze nieujemna.

$$\mathcal{L} \leqslant \log p(x)$$

Wartość dowodu jest stała dla podanego z, więc problem minimalizacji dywergencji można zapisać jako:

$$\operatorname{argmin} D_{KL}(q(z|x)\|p(z|x)) = \operatorname{argmin} \mathcal{L} = \operatorname{argmax} -\mathcal{L} \tag{2.11}$$

2.3.3 Funkcja straty

Korzystając z twierdzenia Bayesa zapisanego w 2.5 można przepisać $\mathcal{L}$ jako:

$$\begin{aligned} \mathcal{L} = \int_z q(z|x) \left(\log \frac{q(z|x)}{p(x,z)} \right) dz = \\ \int_z q(z|x) \left(\log \frac{q(z|x)}{p(x|z)p(z)} \right) dz \end{aligned} \tag{2.12}$$

Wykorzystując kolejny raz własności logarytmów, zapisujemy:

$$\mathcal{L} = \int_z q(z|x) \log \frac{q(z|x)}{p(z)} dz - \int_z q(z|x) \log \frac{q(z|x)}{p(x|z)} dz \tag{2.13}$$

Interpretując poszczególne składniki, $\mathcal{L}$ jest zapisywane jako:

$$\mathcal{L} = D_{KL}(q(z|x)\|p(z)) - \mathbb{E}_{z\sim q(z|x)} \log p(x|z) \tag{2.14}$$

Równanie 2.11 pokazuje, że minimalizacja pierwotnej dywergencji oznacza również minimalizację $\mathcal{L}$.

Pierwsze wyrażenie reprezentuje odległość pomiędzy dystrybucją zastępującą $p(z|x)$, czyli dystrybucją nauczona przez enkoder a $p(z)$, czyli rozkład zmiennych ukrytych, który sami wybierany. W prezentowanym przypadku będzie jest rozkład normalny. Drugie wyrażenie to wartość oczekiwana logarytmu prawdopodobieństwa otrzymania obserwacji x z wartości ukrytych wybieranych z dystrybucji $q(z|x)$, której nauczył się enkoder.

Enkoder, jak i dekoder są zapisywane jako rozkłady prawdopodobieństwa. Funkcja $q(z|x)$ mówi, jakie zmienne ukryte z reprezentują daną x, natomiast $p(x|z)$ generuje dane na podstawie otrzymanego kodu. $q(z|x)$ jest dystrybucją enkodera, a $p(x|z)$ dekodera.

Wariacyjny autoenkoder poruszany w pracy generuje zmienne ukryte z rozkładu normalnego $p(z) = \mathcal{N}(0, \boldsymbol{I})$. Przepuszczając wszystkie nasze dane x przez enkoder, wiemy, że próbkowane wartości przekazywane następnie do dekodera będą rozłożone normalnie. Rozwiązuje to problem tradycyjnego autoenkodera, którego zmienne ukryte są rozrzucone w sposób, który nie jest znany przed treningiem. Znając już dystrybucję, można policzyć dywergencję pomiędzy dwoma rozkładami normalnymi.

$$\frac{1}{2} \sum_{D}^{i=1} \left\{ \left(\frac{\sigma_0}{\sigma_1}\right)^2 + \frac{(\mu_1 - \mu_0)^2}{\sigma_1^2} - 1 + 2 \log \frac{\sigma_1}{\sigma_0} \right\} \tag{2.15}$$

W naszym przypadku, gdzie $\mu_1 = 0$ oraz $\sigma_1 = 1$ uprości się do [1]:

$$\frac{1}{2} \sum_{D}^{i=1} \sigma_i^2 + \mu_i^2 - 2 \log(\sigma_i) - 1 \tag{2.16}$$

gdzie D to długość wektora zmiennych ukrytych.

Jest to pierwsza część funkcji straty.

Druga część funkcji straty jest nazywana błędem rekonstrukcji. Można zastąpić wartość oczekiwaną wartością jednej próbki, ponieważ do trenowania wag modeli używana jest metoda gradientu stochastycznego. Aby uniknąć liczenia skomplikowanych dystrybucji prawdopodobieństwa $\log p(x|z)$, gdzie poszczególne wymiary x są zależne od siebie, liczy się proste rozkłady na każdym wymiarze osobno. Aby to osiągnąć, używa się binarnej entropii krzyżowej. W praktyce często są stosowane również inne funkcje, takie jak błąd średnio-kwadratowy, będący bardziej intuicyjnym rozwiązaniem. Można tak zrobić,

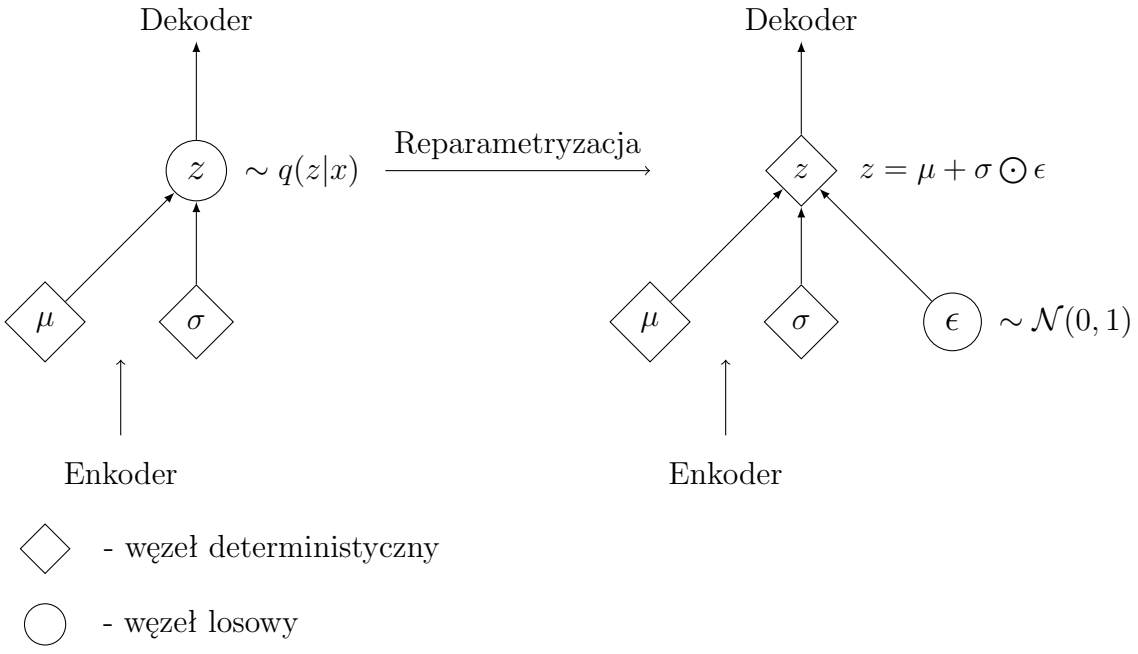
ponieważ wartość prawdopodobieństwa należy do przedziału $[0, 1]$, więc ujemny logarytm z małej liczby (słabego odwzorowania wejścia przez dekoder) będzie ogromna liczbą, a z bliskiej jedynki – małą. W ten sam sposób również zachowuje się średni błąd kwadratowy. Aby lepiej kontrolować trening modelu, można balansować obie części funkcji straty, mnożąc poszczególne wartości przez wybrane stałe, aby model zachowywał się w dokładnie taki sposób jak zostanie to zaplanowane [16].

2.3.4 Metoda reparametryzacyjna

Model *VAE* po zakodowaniu wejścia dokonuje operacji próbkowania (*sampling*) z dystrybucji na nauczonych parametrach. Przy propagacji do przodu nie jest to problem, jednak podczas propagacji wstecznej jest to niemożliwe. Operacja losowania nie jest różniczkowalna, co sprawia, że nie można policzyć gradientu, który jest metodą znajdowania minimalnej wartości funkcji straty. Sposobem obejścia tego problemu jest zastosowanie metody, potocznie nazywanej sztuczką (*reparameterization trick*) [17]. Próbkowanie z dystrybucji $z \sim \mathcal{N}(\mu, \sigma)$ można zapisać jako:

$$\epsilon \sim \mathcal{N}(0, 1)$$
$$z = \mu + \sigma \bigodot \epsilon$$

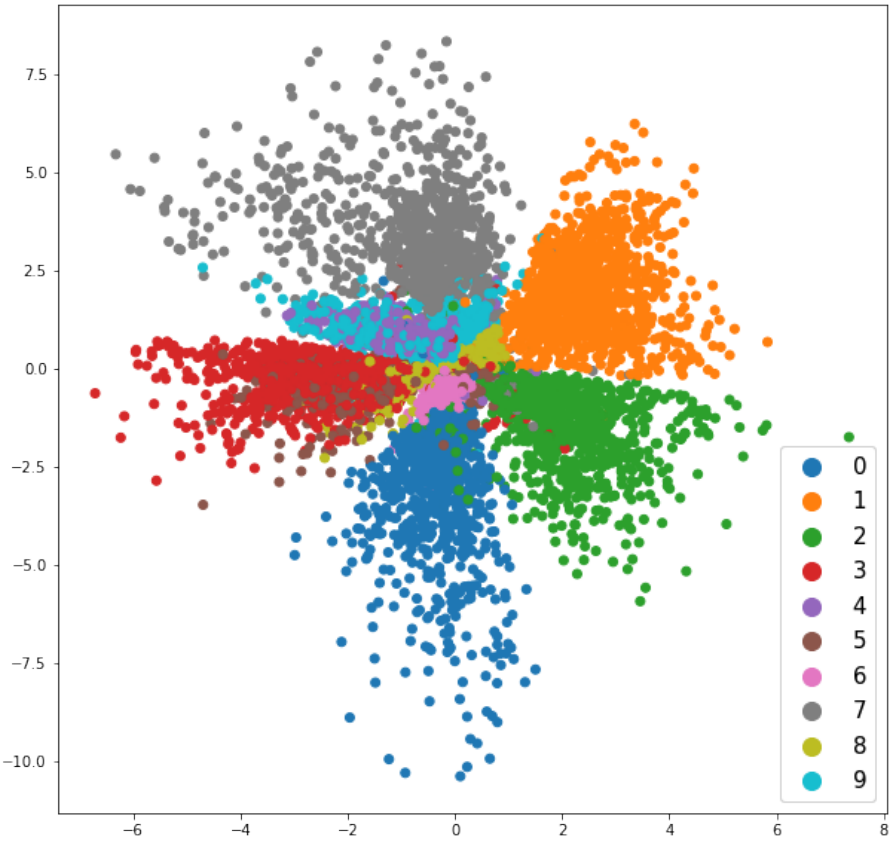
Pozornie nic się nie zmieniło, jednak teraz gradient jest przeprowadzany przez z, idąc od dekodera do enkodera, które jest teraz deterministycznie, co widać na grafice 2.5 [18]. W poprzednim przypadku było ono losowe wybierane z dystrybucji.



Rysunek 2.5: Graficzna reprezentacja reparametryzacji

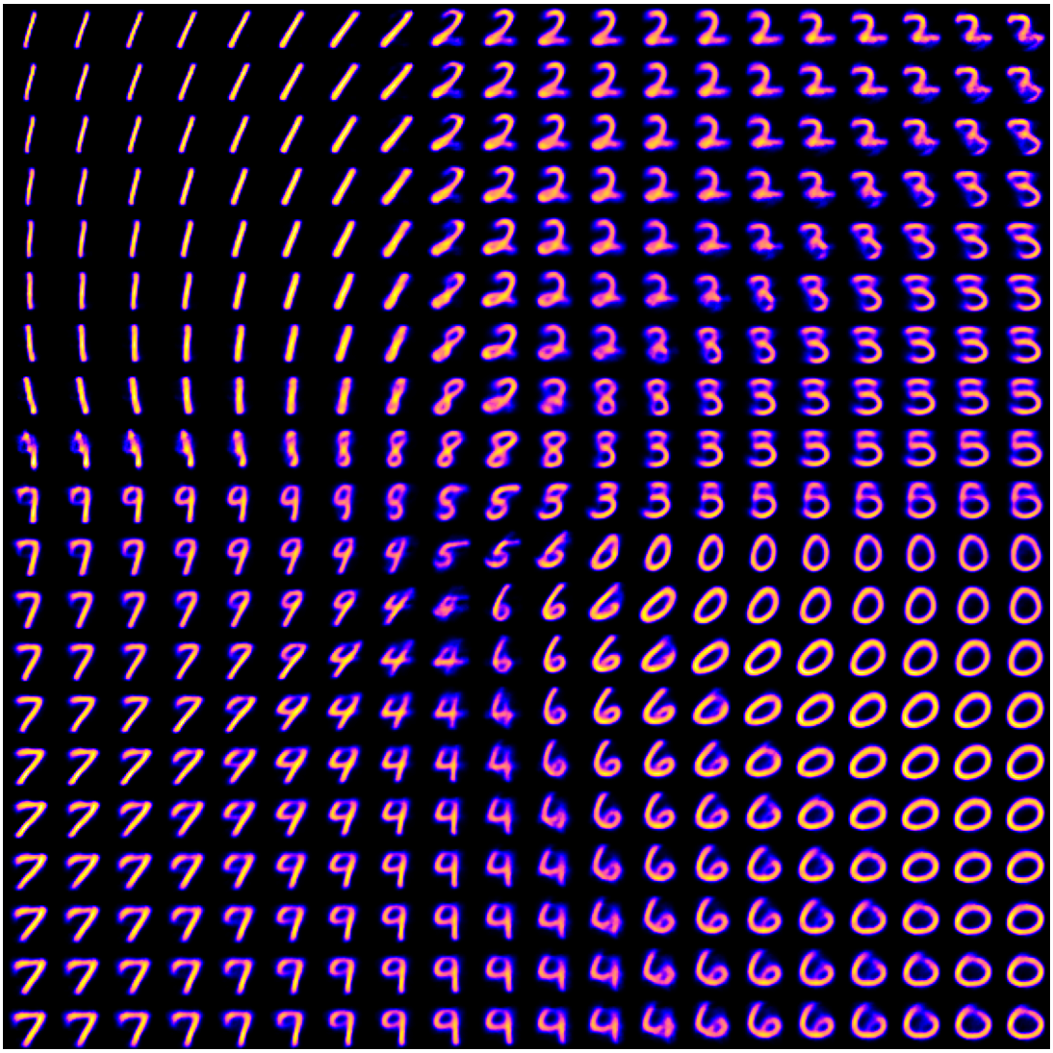
2.4 Zastosowania

Wariacyjny autoenkoder jako pochodna tradycyjnego modelu może zostać zastosowany do takich samych zadań, jak detekcja anomalii, odszumianie oraz kompresja danych. Głównym jednak powodem, dla którego używa się modelu VAE jest jego możliwość generowania danych. Mając już znaną dystrybucję, z której są losowane zmienne ukryte, aby generować nowe dane, podobne do oryginalnych ze zbioru danych. Rysunek 2.6 przestawia zmienne ukryte wygenerowane przez enkoder dla obrazków z MNIST. Porównując obrazek 2.6 z 1.6 jasno widać, że model VAE rozwiązał omówione problemy generacyjne tradycyjnego autoenkodera. Przestrzeń jest kompletna i ciągła. Punkty dystrybucji znajdujące się blisko siebie generują podobne wyniki, co widać na obrazku 2.7. Tak jak chcieliśmy, zmienne ukryte są rozłożone względem rozkładu normalnego.

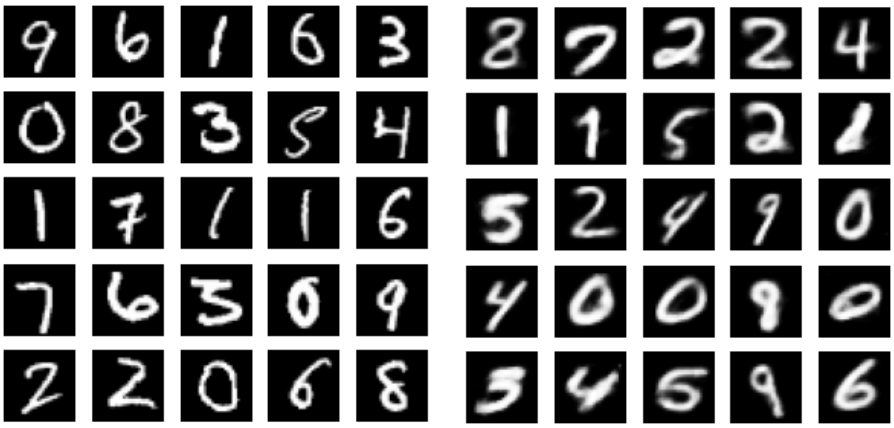


Rysunek 2.6: Przestrzeń zakodowanych zmiennych modelem VAE

Obrazek 2.8 po lewej stronie pokazuje przykładowe dane ze zbioru danych MNIST, a prawa strona pokazuje dane wygenerowane przez wytrenowany model. Generując dane, nie potrzeba już enkodera. Na wejście dekoder dostaje losowo wybrane punkty z dystrybucji, którą wybiera się podczas budowy modelu. W naszym przypadku jest to rozkład normalny.



Rysunek 2.7: Wygenerowane obrazy modelem VAE na przestrzeni zmiennych

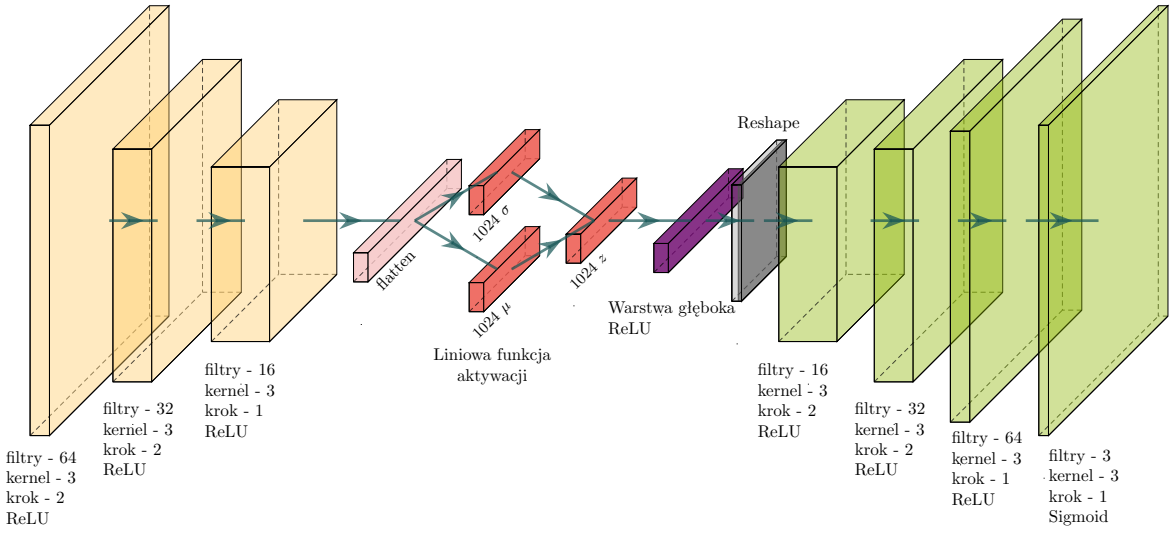


Rysunek 2.8: Porównanie prawdziwych oraz wygenerowanych obrazów

2.4.1 Generacja ludzkich twarzy

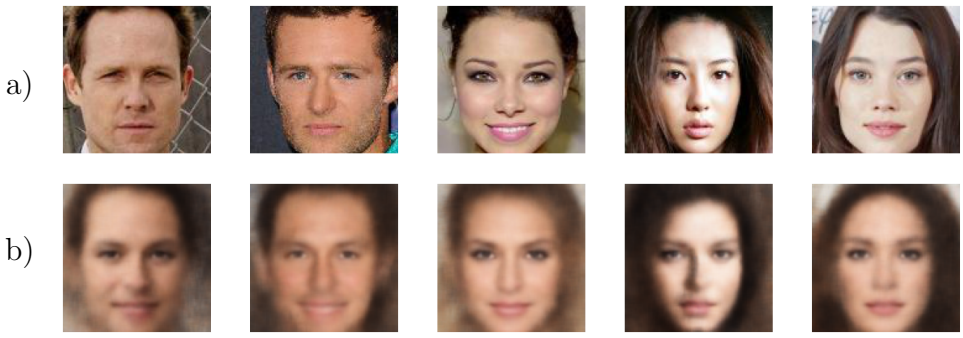
Zbiór danych *CelebFaces Attributes Dataset (CelebA)* jest zbiorem ponad 200 000 twarzy celebrytów opisanych przez 40 atrybutów [19]. Wymiary każdego obrazu to 178 na 218 pikseli. Na podstawie zbioru zostaną zaprezentowane możliwości generacyjne modelu VAE. Obrazom przeznaczonym do treningu sieci, zostały obcięte brzegi, zostawiając obraz o wymiarach 120 na 120 pikseli, co poprawia wydajność modelu poprzez zostawienie tylko twarzy celebtyry bez zbędnych informacji z tła.

Architekturą wykorzystywaną w tym problemie jest konwolucyjny wariacyjny autoenkoder. Sieci konwolucyjne, wchodzące w skład enkodera, wyjątkowo dobrze nadają się do pracy z obrazami [20]. Obrazy zostały zakodowane do rozkładu o wymiarze 1024. Na rysunku 2.9 kolorem żółtym zostały oznaczone warstwy konwolucyjne, a ich poszczególne parametry i funkcje aktywacji zostały opisane pod nimi. W zwężeniu warstwa *flatten* pozwala przejść z trójwymiarowych warstw na jeden wymiar, co umożliwia spłaszczenie obrazów do ukrytych rozkładów. Następnie po operacji losowania w warstwie z, wartości przechodzą do warstwy głębokiej mającej taki sam wymiar jak warstwa *flatten*. *Reshape* zamienia z powrotem wartości jednowymiarowe na trójwymiar, przez co warstwy dekonwolucyjne, oznaczone na zielono, mogą odkodować obraz.



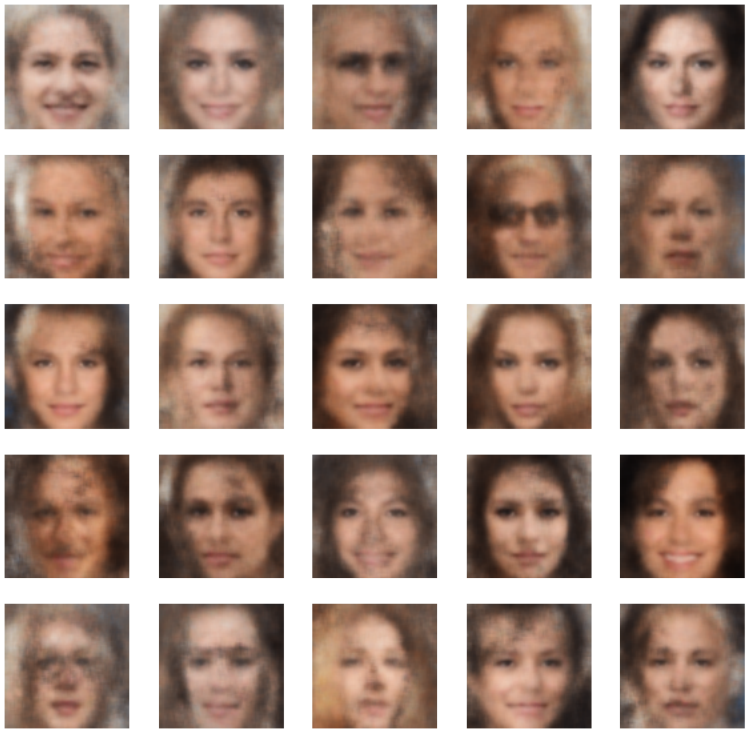
Rysunek 2.9: Architektura modelu konwolucyjnego wariacyjnego autoenkodera

Rysunek 2.10 przedstawia dwa rzędy obrazów, gdzie w rzędzie a są obrazy wchodzące do modelu, a w b ich rekonstrukcje. Jak można zaobserwować, model skupia się na najważniejszych elementach, takich jak usta, oczy oraz nos. Gorzej zrekonstruowanymi elementami są włosy czy tło.



Rysunek 2.10: Rekonstrukcja twarzy modelem VAE

Aby wygenerować nową twarz, wystarczy wylosować punkt z 1024-wymiarowego rozkładu normalnego, a następnie przekazać go na wejście do dekodera.



Rysunek 2.11: Generacja nowych twarzy

Twarze zaprezentowane na rysunku 2.11 są całkowicie nowe. Jak widać na obrazie, model jest w stanie dość dobrze wygenerować podstawowe elementy twarzy jak oczy, nos, usta. W zależności od wylosowanych wartości, twarz może mieć uśmiech, zamknięte usta, różne odcienie koloru skóry lub koloru włosów. Elementy, które nie są obecne na wystarczającej liczbie danych wejściowych takie jak okulary, nie będą dobrze zrekonstruowane przez model. Wynika to niewystarczającej ilości danych treningowych.

2.5 Wnioski

Wariacyjny autoenkoder rozwiązuje problemy generacyjne tradycyjnego modelu. Model VAE implementuje wnioskowanie wariacyjne i dzięki tej metodzie enkoder jest w stanie przybliżyć swoją dystrybucję, aby była jak najbardziej podobna do rozkładu normalnego $\mathcal{N}(0, 1)$. Użycie dywergencji Kullbacka-Leiblera w funkcji straty, zmusza model do nauki wybranego przez nas rozkładu. W ten sposób można wybierać losowo punkty, na podstawie których generowane są nowe dane. Pomimo zmian w sposobie treningu, model dalej jest w stanie być używany do takich samych zastosowań jak tradycyjna architektura, czyli między innymi odszumianie oraz kompresja danych. Budowa modelu jest nietypowa, ponieważ wybierając dystrybucję, trzeba dla każdego parametru charakteryzującego ją, stworzyć warstwę, która uczy się tego konkretnego parametru dla każdego wymiaru kodu. Sprawia to problemy implementacji w wybranej technologii, ponieważ warstwy te muszą być rozłączone pomiędzy sobą, ale jednocześnie połączone z poprzednia i następną warstwą co pokazuje rysunek 2.3 [21].

Rozdział 3

Implementacja

3.1 TensorFlow oraz Keras

3.1.1 Informacje ogóle

TensorFlow jest jedną z najbardziej popularnych bibliotek implementującą metody uczenia maszynowego. Twórcą projektu jest Google, które udostępniło je jako wolne oprogramowanie. Biblioteka jest dostępna dla wielu języków programowania, jednak najczęściej jest używana, programując w języku Python. Programowanie bezpośrednio w TensorFlow jest dość skomplikowane, a kod jest mało czytelny. Problemy te rozwiązuje Keras, który jest nakładką na bibliotekę.

Keras również jest darmową i otwartą biblioteką, jednak jest tylko przeznaczona dla języka Python. Implementuje prosty i przejrzysty sposób tworzenia modeli oraz sieci neuronowych. Poza tradycyjnymi, głębokimi architekturami możliwe jest tworzenie sieci rekurencyjnych oraz konwolucyjnych. Obie biblioteki wspierają rozproszone obliczenia na kartach graficznych.

W implementacji modeli AE i VAE, użyteczną biblioteką jest NumPy. To wolny projekt, ciągle wspierany przez kontrybutorów, umożliwiający operacje na macierzach oraz wektorach. Jako biblioteka napisana w języku C, dokonuje o wiele szybszych operacji w porównaniu do tych, które wykonywane by były w czystym Pythonie.

Istnieje kilka możliwości implementacji tradycyjnego oraz wariacyjnego autoenkodera używając sieci gęstych, konwolucyjnych lub nawet rekurencyjnych takich jak LSTM (*long short-term memory*). Każda z nich może nadać się do innych zbiorów danych, jednak wszystkie w kluczowych punktach działają w ten sam sposób. Wejście zostaje skompresowane do odpowiedniej długości kodu, a w przypadku autoenkoderów wariacyjnych do wielowymiarowego rozkładu normalnego.

3.1.2 Implementacja w języku Python

Potrzebne klasy i pakiety są importowane w listingu 3.1.

```
from tensorflow.keras.layers import Input, Flatten, Dense,
    Lambda, Reshape, Layer
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.datasets import mnist
import tensorflow.keras.backend as K
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
```

Listing 3.1: Importy klas i funkcji

Następnie wczytany zbiór danych MNIST, jest dzielony na część przeznaczoną do treningu oraz do testów. Modele AE i VAE nie korzystają z oznaczeń, do jakiej klasy należą poszczególne dane, ponieważ oba modele na swoją warstwę wejściową, jak i wyjściową dostają te same obrazy. Aby nie było problemów z typem danych, najlepiej zamienić wszystkie próbki na macierze, które przechowują dane w formacie float o długości 32 bitów. Następnie dane są skalowane do przedziału [0, 1], dzieląc każdy piksel każdego obrazka przez 255. Dane składają się z obrazów w skali szarości i każdy piksel opisuje liczba z przedziału [0, 255], więc zwykłe dzielenie może zastąpić inne rozwiązania skalowania danych. Zapisano do zmiennych wysokość i szerokość obrazów, aby prościej się do nich odwoływać.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train / 255
x_test = x_test / 255

szerokosc = x_train[0].shape[0]
wysokosc = x_train[0].shape[1]
```

Listing 3.2: Przygotowanie zbioru danych

Autoenkoder

W listingu 3.3 tworzony jest enkodera z dwiema warstwami ukrytymi, z czego pierwsza posiada 500 neuronów, a druga 120. Obraz jest kompresowany do kodu o długości 2. Architektura dekodera jest odbiciem lustrzanym enkodera. Funkcje aktywacji warstw ukrytych to ReLU *Rectified Linear Unit*, które nie dość, że umożliwiają lepszą naukę sieci, to sprawiają, że trening zajmuje mniej czasu [22]. Użyta funkcja liniowa w warstwie kodu, nie ogranicza wartości, jakie poszczególne jego elementy mogą przyjąć.

Funkcja sigmoidalna na wyjściu spłaszcza wyjście neuronu do przedziału [0, 1], czyli takiego samego, w jakim występują nasze dane.

Pierwsza warstwa wejściowa jest przeznaczona, aby przyjmować dane o wymiarze takim jak obrazy. Następnie warstwa *Flatten* zamienia dwuwymiarowe wejście na jednowymiarowe. Kolejne warstwy łączą się w normalny sposób pomiędzy sobą. Ostatnia warstwa *Reshape* dokonuje odwrotnej operacji co *Flatten*, zamieniając jednowymiarowe wartości na macierz, która ma takie same wymiary co dane wejściowe, co umożliwia porównanie wyniku modelu do wejścia. Do stworzenia całego autoenkoder,a używa się klasy *Model*, podając pierwszą oraz ostatnią warstwę. W ten sposób wszystkie parametry są trenowane na raz. Stworzenie osobno enkodera nie jest problemem, ponieważ posiada on tą samą warstwę wejściową co cały model. Dekoder jest nieco problematyczny, ponieważ należy stworzyć jego własną warstwę wejściową, a następnie połączyć z nią warstwy modelu, tak aby korzystał z wytrenowanych parametrów.

```
dlugosc_kodu = 2
wejscie = Input(shape=(szerokosc,wysokosc))
x = Flatten()(wejscie)
x = Dense(500, activation='relu')(x)
x = Dense(120, activation='relu')(x)
kod = Dense(dlugosc_kodu, activation='linear')(x)
x = Dense(120, activation='relu')(kod)
x = Dense(500, activation='relu')(x)
x = Dense(szerokosc * wysokosc, activation='sigmoid')(x)
wyjscie = Reshape([szerokosc, wysokosc])(x)

autoenkoder = Model(wejscie, wyjscie)
enkoder = Model(wejscie, kod)
dekoder_wejscie = Input(shape=(dlugosc_kodu, ))
dec_1 = autoenkoder.layers[5]
dec_2 = autoenkoder.layers[6]
dec_3 = autoenkoder.layers[7]
dec_4 = autoenkoder.layers[8]

decoder = Model(dekoder_wejscie, dec_4(dec_3(dec_2(dec_1(
    dekoder_wejscie)))))
```

Listing 3.3: Stworzenie autoenkodera

Zanim trening sieci się rozpocznie, trzeba sprecyzować funkcję straty oraz optymalizator. Funkcja użyta w przykładzie 3.4 to błąd średnio-kwadratowy, który jest dobrym wyborem w przypadku porównywania obrazów. Optymalizator to algorytm znajdowania wag sieci, dla których funkcja straty jest jak najmniejsza, a co za tym idzie model działa jak najlepiej. Wybraną metodą jest algorytm *adam*, który jest pochodną stochastycznego gradientu. Obliczenia okupują mniej pamięci, jest prosty obliczeniowo oraz znajduje

optymalne rozwiązanie szybciej niż tradycyjne podejście [23]. Jednym z jej twórców jest Diedrik Kingma, który również jako pierwszy zaproponował model wariacyjnego autoenkodera. Aby nie przetrenować modelu, dodaje się zbiór walidacyjny, który przejmuje 20% danych treningowych. Walidacja służy do sprawdzenia na innych danych niż testowe i treningowe, czy model się nie przetrenował. W momencie, kiedy błąd na zbiorze walidacyjnym będzie rosnąć przez 3 epoki zamiast maleć, callback *EarlyStopping* przerwie trening wag i przywróci wagi, dla których wynik był najlepszy.

```
stop = EarlyStopping(restore_best_weights=True, patience=3)

autoenkoder.compile(optimizer='adam', loss='mse')
autoenkoder.fit(x_train, x_train, epochs=100,
    validation_split = .2, callbacks=[stop])
```

Listing 3.4: Trening modelu

Po treningu modele są już gotowe go użytkowania. Aby dokonać przy ich pomocy obliczeń, należy wykonać funkcję *predict*, do której trzeba podać odpowiednich wymiarów macierz.

Wariacyjny autoenkoder

Przygotowanie zbioru danych dla wariacyjnego autoenkodera nie różni się w żaden sposób od pokazanego w 3.2.

Implementacja modelu VAE jest bardziej skomplikowana niż AE. Bierze się to jego budowy, która wymaga rozłączenia jednej warstwy na dwie niezależne przyłączone do tej samej, co widać na obrazku 2.3 oraz implementacji własnej funkcji straty. Warstwy reprezentujące średnią i odchylenie standardowe są połączone z ostatnią warstwą ukrytą. W listingu 3.5 warstwę odchyleń standardowych traktuje się jako ich logarytm, ponieważ jego wartość nie może być ujemna. Kiedy potrzeba użyć zmiennych, wystarczy dokonać na nich operacji *exp*. Funkcja *probka* dokonuje losowania z dystrybucji na nauczonych parametrach, stosując opisaną wcześniej metodę reparametryzacyjną. Zmienna *eps* przechowuje wartości losowane z rozkładu normalnego i ich rozmiar jest równy ilości zmiennych ukrytych i wielkości partii danych (*batch size*). Na etapie tworzenia architektury modelu nie wiadomo o wielkości batch-a, więc używając metody *shape* można ją dynamicznie zmieniać w zależności od przekazanych parametrów. Dostępna w paczce Keras klasa *Lambda* pozwala wywołać wskazaną funkcję na wartościach neuronów. Warstwa jest łączona z obiema wcześniejszymi warstwami na raz. Implementacja dekodera jest identyczna jak dla tradycyjnego autoenkodera pokazanego w 3.3. Funkcja straty modelu VAE nie jest możliwa do automatycznego policzenia, tak jak zostało to pokazane w AE. Zmienna *z_odkodowane* przechowuje obraz odkodowany ze zmiennych ukrytych, który będzie porównywany z tym przekazanym na wejście sieci.

```
n_ukrytych = 2
enkoder_wejscie = Input(shape=(szerokosc, wysokosc))
x = Flatten()(enkoder_wejscie)
x = Dense(500, activation='relu')(x)
x = Dense(120, activation='relu')(x)

mu = Dense(n_ukrytych)(x)
log_sigma = Dense(n_ukrytych)(x)

def probka(args):
  i_mu, i_log_sigma = args
  eps = K.random_normal(shape=(K.shape(i_mu)[0],
                               K.shape(i_mu)[1]))
  return i_mu + K.exp(i_log_sigma) * eps

z = Lambda(probka,output_shape=(n_ukrytych,))([mu,log_sigma])
enkoder = Model(enkoder_wejscie, [mu, log_sigma, z])

dekoder_wejscie = Input(shape=(n_ukrytych, ))
x = Dense(120, activation='relu')(dekoder_wejscie)
x = Dense(500, activation='relu')(x)
x = Dense(szerokosc * wysokosc, activation='sigmoid')(x)
dekoder = Model(dekoder_wejscie, x)

z_odkodowane = dekoder(z)
```

Listing 3.5: Stworzenie modelu wariacyjnego autoenkodera

```
class WarstwaStraty(Layer):

  def vae_loss(self, x, z_odkodowane):
    x = K.flatten(x)
    z_odkodowane = K.flatten(z_odkodowane)
    blad_rekonstrukcji = K.sum(K.square(x-z_odkodowane))
    kld = -0.5 * K.sum(1 + 2 * log_sigma - K.square(mu)
          - K.square((K.exp(log_sigma))), axis=-1)
    return K.mean(blad_rekonstrukcji + kld)

  def call(self, inputs):
    x = inputs[0]
    z_odkodowane = inputs[1]
    loss = self.vae_loss(x, z_odkodowane)
    self.add_loss(loss, inputs=inputs)
    return x

y = WarstwaStraty()([enkoder_wejscie, z_odkodowane])
```

Listing 3.6: Obliczenie warstwy straty VAE

Aby policzyć wartość funkcji straty, można stworzyć nową warstwę przyłączoną do ostatniej. Jej jedynym zadaniem jest policzenie straty i nie ma ona możliwych do treno-

wania parametrów. Jej budowa została zaprezentowana na listingu 3.6. Pierwsza metoda w klasie oblicza wartość funkcji straty, a druga pozwala jej policzenie i minimalizacje przez model. Aby móc porównać obrazy, muszą być one w takich samych wymiarach, co gwarantuje wywołanie funkcji *flatten* na prawdziwym i odkodowanym obrazie. Błąd rekonstrukcji wybrany w tym przykładzie to błąd średnio-kwadratowy. Zmienna *kld* przechowuje obliczoną wartość dywergencji Kullbacka-Leiblera z wyprowadzonego wzoru 2.16. Warstwa obliczająca stratę jest połączona z wejściem i wyjściem całego modelu, co umożliwia dostęp do wejścia ze zbioru danych oraz obrazów odkodowanych ze zmiennych ukrytych.

```
1 stop = EarlyStopping(restore_best_weights=True, patience=3)
2 vae = Model(enkoder_wejscie, y)
3
4 vae.compile(optimizer='adam', loss=None)
5
6 vae.fit(x_train, None, epochs = 100, batch_size = 32,
     validation_split = 0.2, callbacks=[stop])
7
```

Listing 3.7: Trening modelu

Posiadając warstwę obliczającą funkcję straty, można stworzyć model. Podczas kompilacji nie należy podawać parametru *loss*, ponieważ został on już dodany w niestandardowej warstwie. Z tego samego powodu w funkcji mającej dokonać treningu modelu nie przekazuje się danych, do których jest porównywany wynik autoenkodera.

Rozdział 4

Podsumowanie

Autoenkoder tradycyjny oraz wariacyjny są podobnymi modelami uczenia maszynowego. Składają się z dwóch części: enkodera próbującego zakodować zmienne do kodu o określonej długości oraz dekodera rekonstruującego kod do danych wejściowych. Wariacyjny autoenkoder, zamiast generować zmienne bezpośrednio, wybiera je z wielowymiarowego rozkładu normalnego. Jest to sposób na rozwiązanie problemów z generacją nowych danych tradycyjnego modelu. Oba modele posiadają szerokie zastosowanie w kompresji oraz odszumianiu danych i detekcji anomalii. Przewaga wariacyjnego autoenkodera polega na jego umiejętności generacji danych, podobnych do tych, na których on został wytrenowany.

Spis rysunków

Spis Listingów

Bibliografia

[1] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

[2] Adam Roberts et al. *A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music*. 2019. arXiv: 1803.05428 [cs.LG].

[3] Dor Bank, Noam Koenigstein, and Raja Giryes. *Autoencoders*. 2021. arXiv: 2003.05991 [cs.LG].

[4] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[5] Izaak Neutelings. *Neural networks*. URL: https://tikz.net/neural_networks/.

[6] Matus Telgarsky. *Benefits of depth in neural networks*. 2016. arXiv: 1602.04485 [cs.LG].

[7] Ronen Eldan and Ohad Shamir. *The Power of Depth for Feedforward Neural Networks*. 2016. arXiv: 1512.03965 [cs.LG].

[8] Jarosław Gramack and Artur Gramacki. *Wybrane metody redukcji wymiarowości danych oraz ich wizualizacji*. 2008.

[9] Saïd Ladjal, Alasdair Newson, and Chi-Hieu Pham. *A PCA-like Autoencoder*. 2019. arXiv: 1904.01277 [cs.CV].

[10] Herminarto Nugroho. "Fully Convolutional Variational Autoencoder For Feature Extraction Of Fire Detection System". In: *Jurnal Ilmu Komputer dan Informasi* 13 (Mar. 2020). DOI: 10.21609/jiki.v13i1.761.

[11] Elad Plaut. *From Principal Subspaces to Principal Components with Linear Autoencoders*. 2018. arXiv: 1804.10253 [stat.ML].

[12] Christian Versloot. *What is a Variational Autoencoder (VAE)?* 2020. URL: https://www.machinecurve.com/index.php/2019/12/24/what-is-a-variational-autoencoder-vae/.

[13] Tim Salimans, Diederik P. Kingma, and Max Welling. *Markov Chain Monte Carlo and Variational Inference: Bridging the Gap*. 2015. arXiv: 1410.6460 [stat.CO].

[14] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. "Variational Inference: A Review for Statisticians". In: *Journal of the American Statistical Association* 112.518 (2017), 859–877. ISSN: 1537-274X. DOI: 10.1080/01621459.2017.1285773. URL: http://dx.doi.org/10.1080/01621459.2017.1285773.

[15] Frank Noe. *Deep Learning Lecture 11.2 - Variational Inference*. Youtube. 2020. URL: https://www.youtube.com/watch?v=IkxQxdYSmrM.

[16] Andrea Asperti and Matteo Trentin. *Balancing reconstruction error and Kullback-Leibler divergence in Variational Autoencoders*. 2020. arXiv: 2002.07514 [cs.NE].

[17] Ming Xu et al. *Variance reduction properties of the reparameterization trick*. 2018. arXiv: 1809.10330 [stat.ML].

[18] Diederik P. Kingma and Max Welling. "An Introduction to Variational Autoencoders". In: *Foundations and Trends® in Machine Learning* 12.4 (2019), 307–392. ISSN: 1935-8245. DOI: 10.15612200000056. URL: httpdx.doi.org10.15612200000056.

[19] Ziwei Liu et al. "Deep Learning Face Attributes in the Wild". In: *Proceedings of International Conference on Computer Vision (ICCV)*. 2015.

[20] Yunchen Pu et al. "Variational Autoencoder for Deep Learning of Images, Labels and Captions". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/eb86d510361fc23b59f18c1bc9802cc6-Paper.pdf.

[21] Janosh Riebesell. *VAE*. URL: https://tikz.netlify.app/vae.

[22] Antoine Bordes Xavier Glorot and Yoshua Bengio. *Deep Sparse Rectifier Neural Networks*. 2011.

[23] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].