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. 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 pobabilistyczne podejście do generowania zmiennych ukrytych. Swoją popularność zawdzięcza swojej budowie, która jest oparta na sieciach neuronowych oraz możliwości trenowania ich przy pomocy metod gradientowych.

Rozdział 1

Tradycyjny autoenkoder

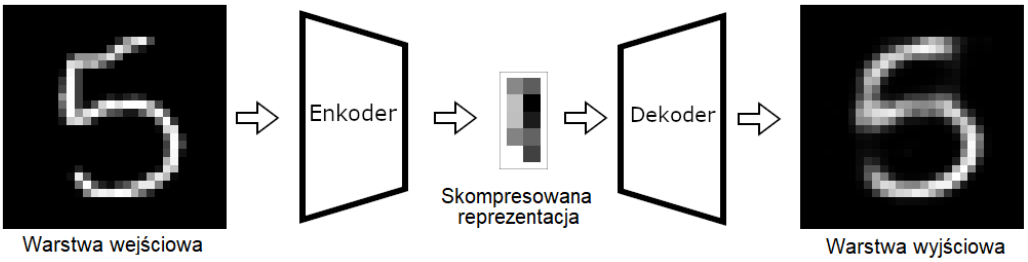
1.1 Informacje wstępne

Autoenkoder jest specyficzną wersją sieci neuronowej składającej się z dwóch części: enkodera, który koduje dane wejściowe oraz dekodera, który na podstawie kodu rekonstruuje wejście.[2] Architektura enkodera wymaga aby jego warstwa wyjściowa generująca reprezentacje danych była mniejsza niż warstwa wejściowa. Często zwężenie to jest nazywane *bottle neck*. Model na swoją warstwę wejściową oraz wyjściową dostaje te same dane. Powiedzmy że mamy dane wejściowe X o wymiarze m oraz chcemy je zakodować do wymiaru n. Formalnie możemy zapisać:

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

Celem *bottle neck-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 lub binarna entropia krzyżowa, która powie nam, jak wynik różni się od wejścia.

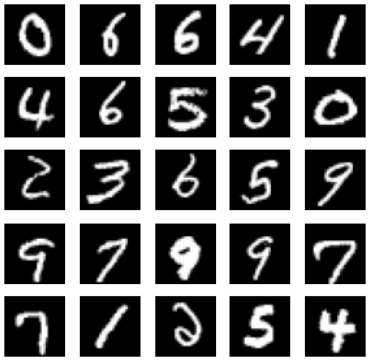
$$\mathcal{L}(x, \hat{x}) = \frac{1}{m}\sum_{i=0}^{m}(x_i - \hat{x}_i)^2 = \frac{1}{m}\sum_{i=0}^{m}(x_i - D(E(x_i)))^2$$



Rysunek 1.1: Schemat budowy autoenkodera.

1.1.1 Zbiór danych MNIST

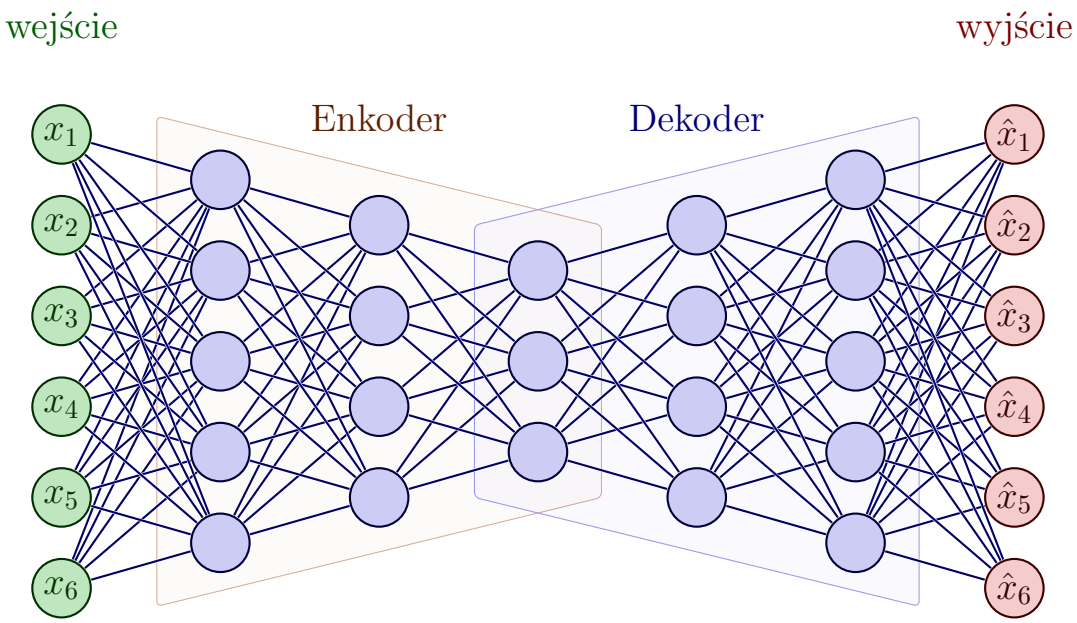
Zbiór danych MNIST (*Modified National Institute of Standards and Technology*) jest zbiorem wielu odręcznie pisanych cyfr.[3] Znajduje szerokie zastosowanie w nauce i prezentacjach możliwości modeli uczenia maszynowego. W jego skład wchodzi 60,000 obrazów przeznaczonych do treningu modeli oraz 10,000 do testów. Obrazy są czarno-białe i mają wymiary 28 na 28 pikseli.



Rysunek 1.2: Przykładowe obrazy ze zbioru danych.

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 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 możemy ustalić przed jego treningiem to:

- Ilość warstw ukrytych - jeśli wiemy, 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.[4, 5]

- 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 dla nas i dla modelu. Dłuższy kod oznacza zachowanie więcej istotnych elementów, a co za tym idzie lepsze odwzorowanie przez dekoder. Z drugiej strony używając, dłuższego wektora zmiennych ukrytych dostajemy gorszą kompresje danych. Długość kodu musimy dobrać w zależności od problemu, który chcemy rozwiązać używając model.

1.3 Zastosowania

1.3.1 Redukcja wymiaru

Żyjemy w świecie, który otacza nas coraz większą ilością informacji. Jej ilość sprawia, że możemy z jej wyciągać coraz to 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 chcemy zmniejszyć wymiarowość danych 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.

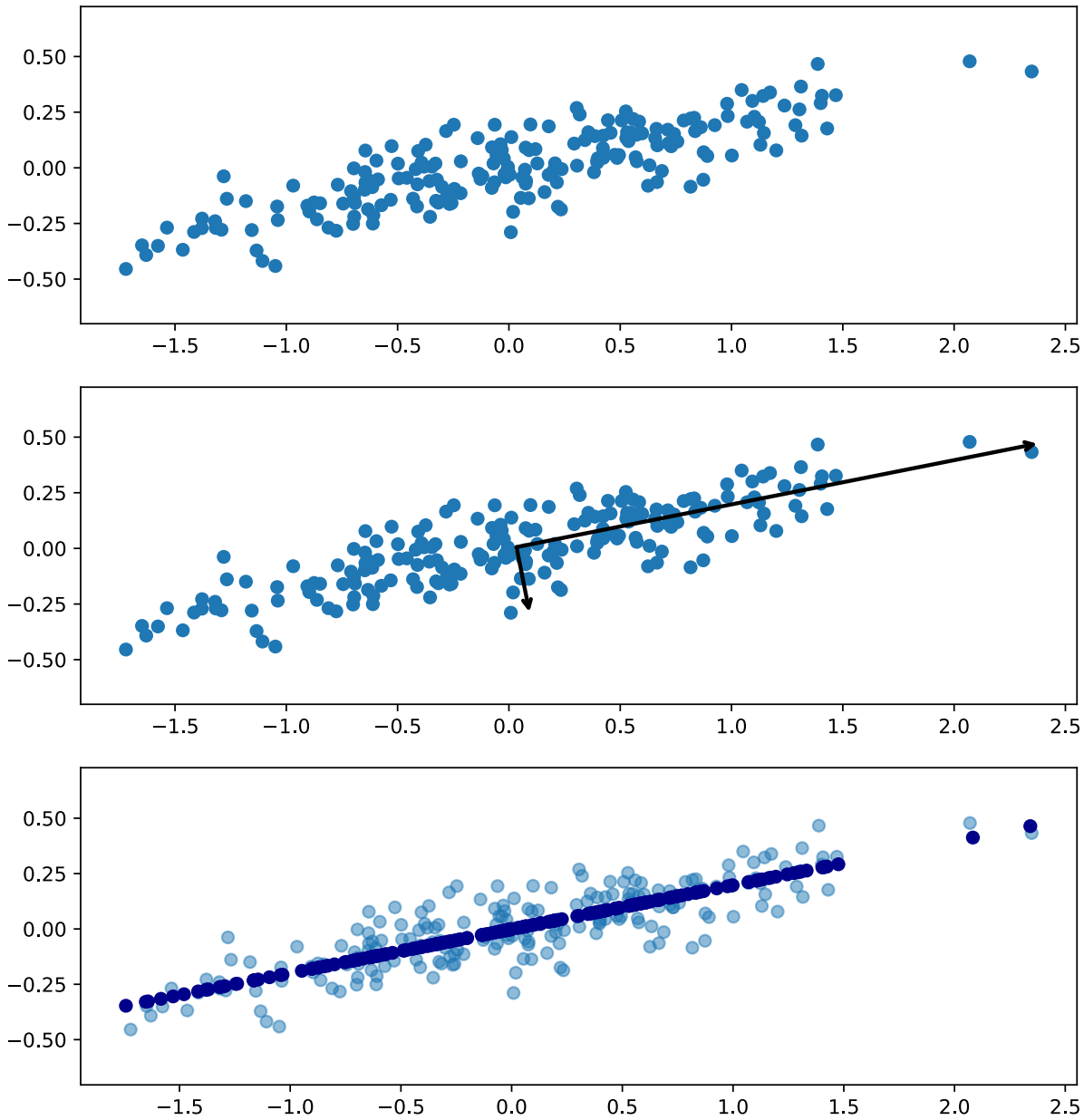
- Wielowymiarowe dane jest ciężej zwizualizować. Możemy je zredukować do jedno, dwu, lub trzy wymiarowej reprezentacji co pozwoli nam na proste narysowanie wykresu, który będzie prosty do zrozumienia.

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

Analiza składowych głównych

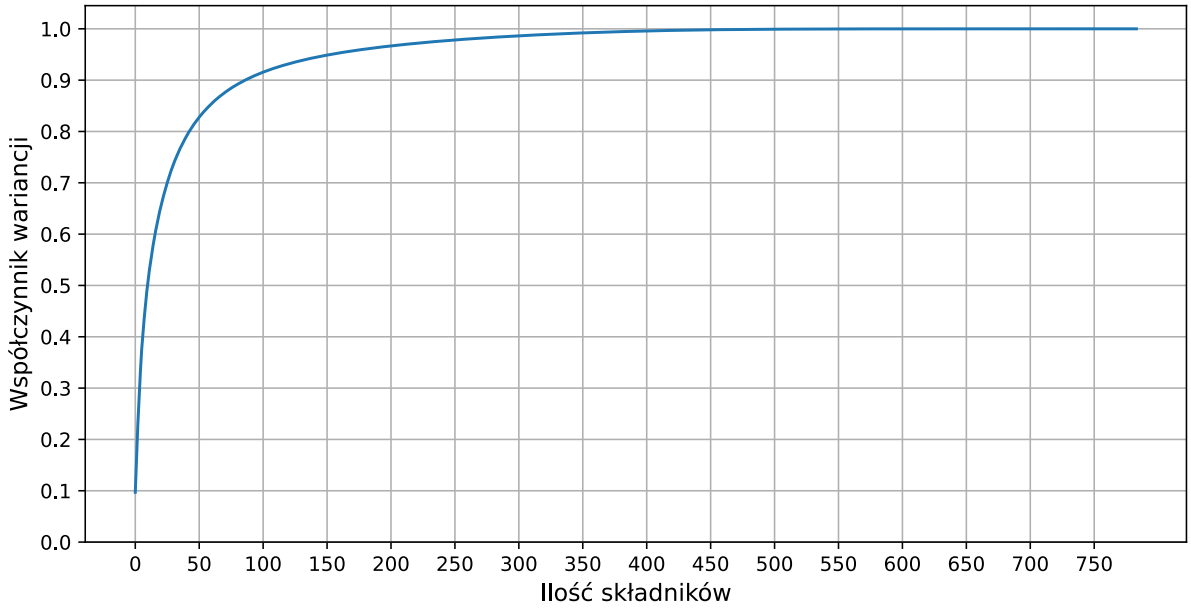
Analiza składowych głównych (*ang. Principal Components Analysis,* ***PCA***) służy do wyznaczania jak najmniejszej ilości nowych zmiennych, mówiących jak najwięcej o zbiorze danych. Wielowymiarowe dane koncentrują się w pewnych podprzestrzeniach oryginalniej przestrzeni. Analiza PCA pozwala znaleźć te podprzestrzenie, które są wektorami pełniącymi rolę nowych osi, które je lepiej opisują nasz zbiór danych.[6] Używając tej metody ograniczamy się tylko do przekształceń liniowych. Ilość wektorów względem których można zredukować dane jest równa wymiarowi danych. Środkowy obrazek 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. Dolny wykres rysunku 1.4 pokazuje już spłaszczone 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 osiągamy praktycznie 100% pokrycia.



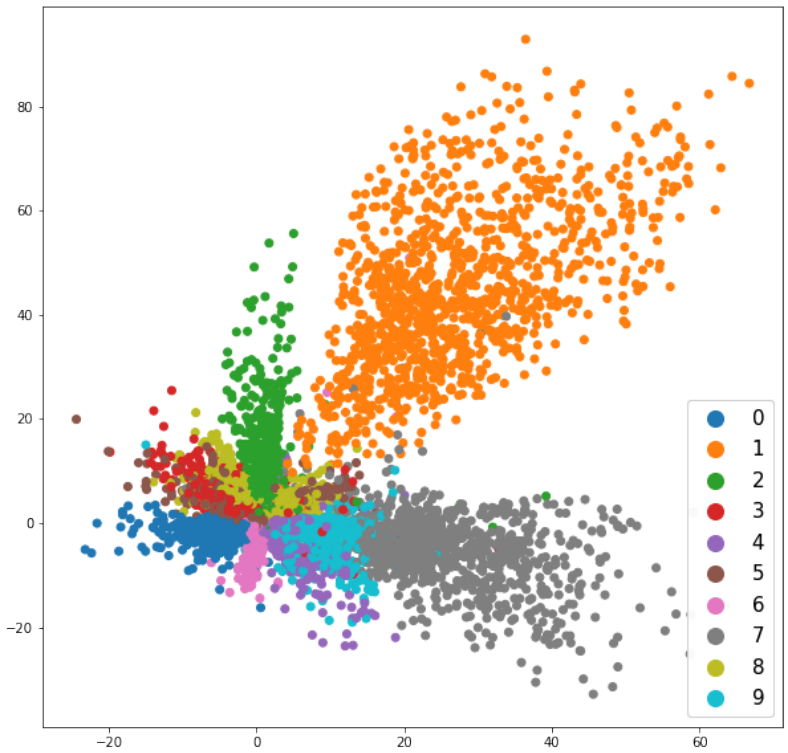
Rysunek 1.5: Procent wariancji opisywany przez ilość składników.

Autoenkoder

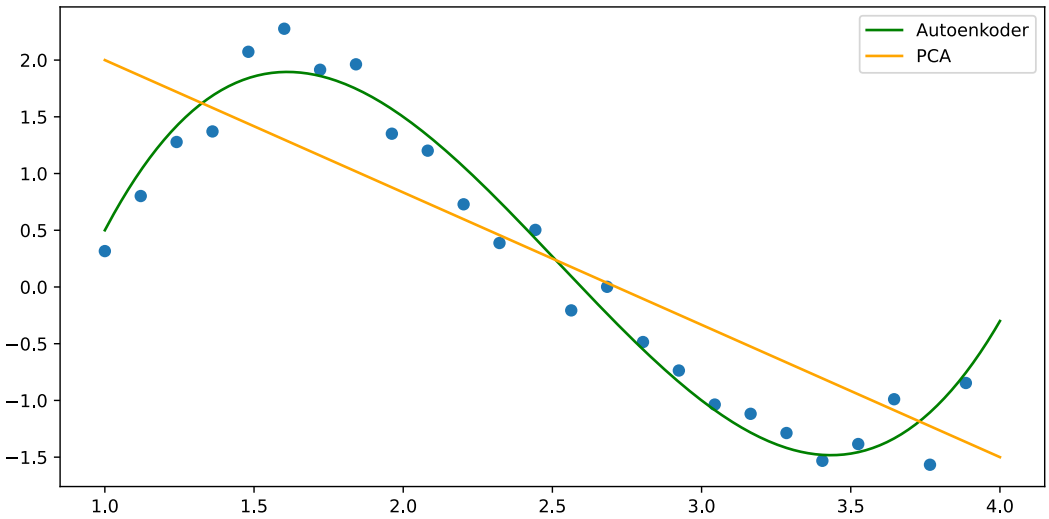
Użycie autoenkodera jest jedną z możliwości jaką mamy jeśli chcemy dokonać redukcji wymiarów. Jego budowa, wymusza naukę jak najlepszej reprezentacji danych w kodzie. Zadaniem enkodera jest zachowanie w zmiennych jak najwięcej informacji dotyczących wejścia, a dekoder ma z nich odwzorować jak najwięcej informacji. Najważniejszą cechą autoenkodera jest możliwość nauki przekształceń zarówno liniowych jak i nieliniowych w zależności od doboru funkcji aktywacji.

Porównanie obu metod

Obie przedstawione metody redukowania wymiarów są bardzo dobrymi wyborami. Wybór jednej z nich powinien zostać dopasowany do naszych potrzeb i do zbioru danych którego używamy. PCA, będąc ograniczone do jedynie liniowych przekształceń jest szybsze niż autoenkoder, który wymaga treningu. Jest ono też lepszym wyborem kiedy wiemy, że nasze dane są nadmiernie skorelowane. Rozpatrzmy przykład zbioru danych opisującego takie cechy ludzi jak wzrost, waga, kolor oczu oraz długość włosów. Wiemy, że czyjaś waga jest zależna od wzrostu. PCA w takim przykładzie bez problemu odnajdzie tą zależność i zredukuje. Autoenkoder jest lepszym wyborem, kiedy mamy do czynienia z danymi, które są o wiele bardziej złożone, czyli obrazy czy pliki audio. Różnica pomiędzy nauką liniowych oraz nielinowych przekształceń została pokazana na rysunku 1.3.1. Jednowarstwowy autoenkoder z liniową funkcja aktywacji na każdej warstwie zachowuje się dokładnie tak samo jak PCA.[7]



Rysunek 1.6: Przestrzeń dwuwymiarowej zmiennej ukrytej dla zbioru MNIST.

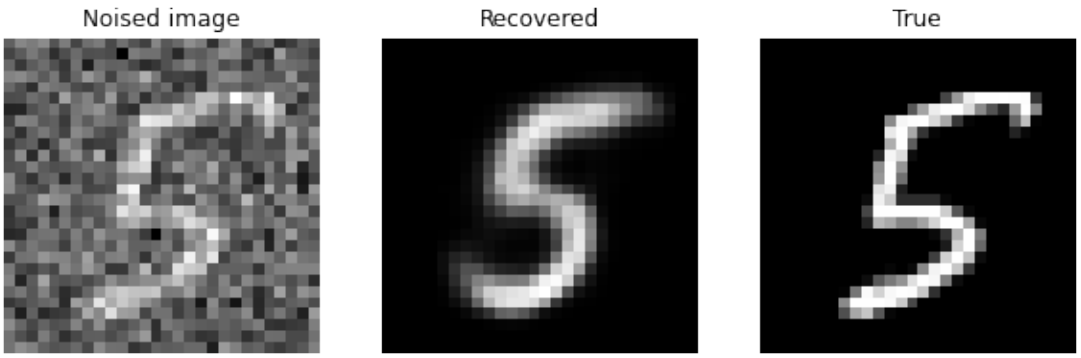


Rysunek 1.7: Liniowe i nieliniowe przekształcenie.

1.3.2 Odszumianie obrazów

Model autoenkodera przeznaczony do odszumiania danych, często dostaje swoją nazwę i jest określany mianem DAE *(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 warstwy 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 naszym głównym zadaniem jest jak najlepsze odtworzenie danych wejściowych jest to kłopot, jednak w tym przypadku możemy wykorzystać tą własność na naszą korzyść. Porównując wyjście modelu z danymi bez szumu, zapewniamy, że model nauczy się w jakimś stopniu odtwarzać je poprawnie. Zmuszamy w ten sposób model do ignorowania nieistotnych części naszych 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 do treningu został dodany szum. Zaszumiony obraz powstał przez dodanie do oryginalnego obrazu losowo wybranych wartości z rozkładu Gaussa $\mathcal{N}(0, 1)$ przemożonych przez stałą, która w tym przypadku wynosi 0.4. Następnie obrazy wejściowe zostały skompresowane do kodu o długości pięć, a następnie odkodowane przez dekoder.



Rysunek 1.8: Obraz z szumem, odzyskany oraz prawdziwy.

Wadą autoenkoderu 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 się nadawał do innego zbioru, z którego danymi będziemy chcieli pracować. Jedynym rozwiązaniem tego problemu jest stworzenie nowego modelu przeznaczonego do użytku na nowych danych.

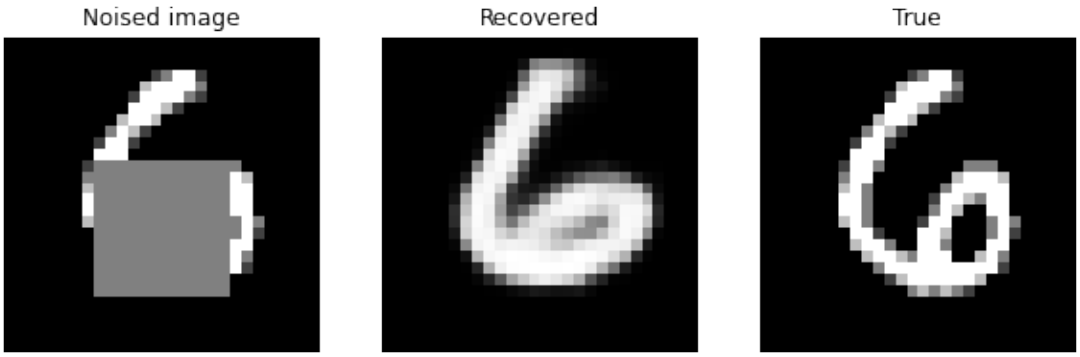
1.3.3 Uzupełnianie obrazów

Uzupełnianie obrazów ma na celu wypełnienie brakującej lub zamaskowanej części obszaru. Człowiek bez problemu jest sobie poradzić z tym zadaniem 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 jesteśmy generować nowe dane podobne do tych, na których model został wytrenowany. Wiemy, że sieć po treningu, jest w stanie ze zmiennych ukrytych odkodować obraz, więc ustawiając wejście dekodera na losowy punkt z przestrzeni zmiennych powinniśmy być w stanie 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

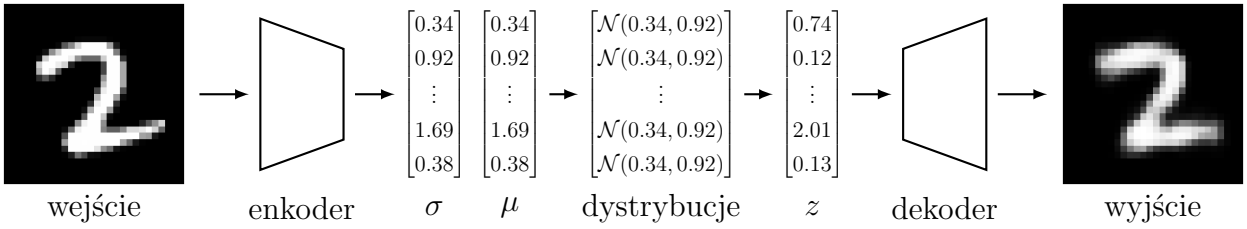
Tradycyjna architektura nie zapewnia nam przed treningiem, żadnego z tych warunków. Spoglądając na rysunek 1.6 możemy zaobserwować, że przestrzeń zawiera luki. Szczególnie dobrze to widać między klasami oznaczającymi jedynki oraz siódemki. Kolejnym problemem widocznym na tej grafice jest brak separacji 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 nasze warunki. Może się tak zdarzyć, że sieć nauczy się akurat takiej dystrybucji, która nam pasuje, ale jest to bardzo mało prawdopodobne. Jeśli chcemy zbudować model generacyjny musimy mieć zagwarantowane, że za każdym razem dostaniemy rozkład spełniający odpowiednie warunki.

Rozdział 2

Wariacyjny autoenkoder

2.1 Informacje ogóle

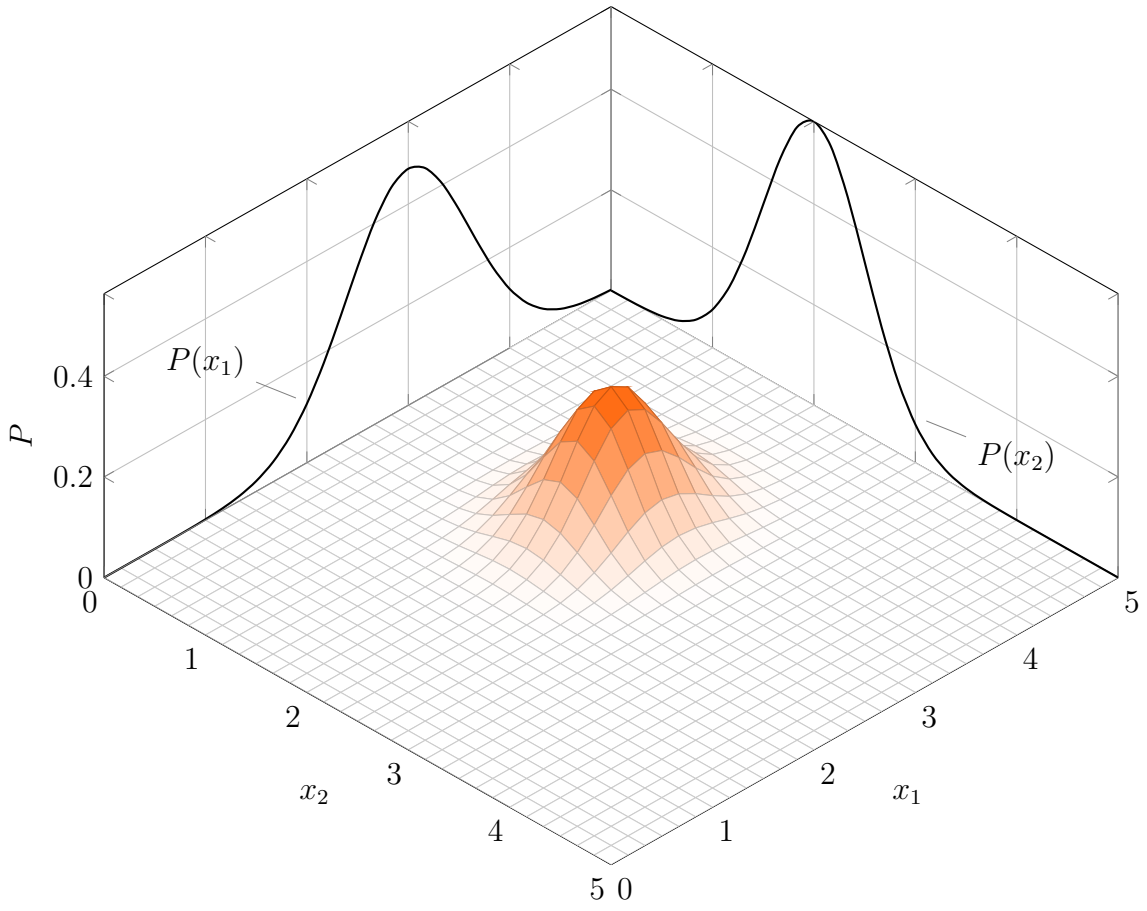
Wariacyjny autoenkoder rozwiązuje problemy generacyjne tradycyjnego modelu. VAE ma na celu skompresowanie danych do określonego wielowymiarowego rozkładu ukrytego, a następnie z próbki tej dystrybucji, próbuje jak najlepiej zrekonstruować wejście. Model ten, należy go grupy wariacyjnych metod Bayesowskich, 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 chcemy dane skompresować 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.



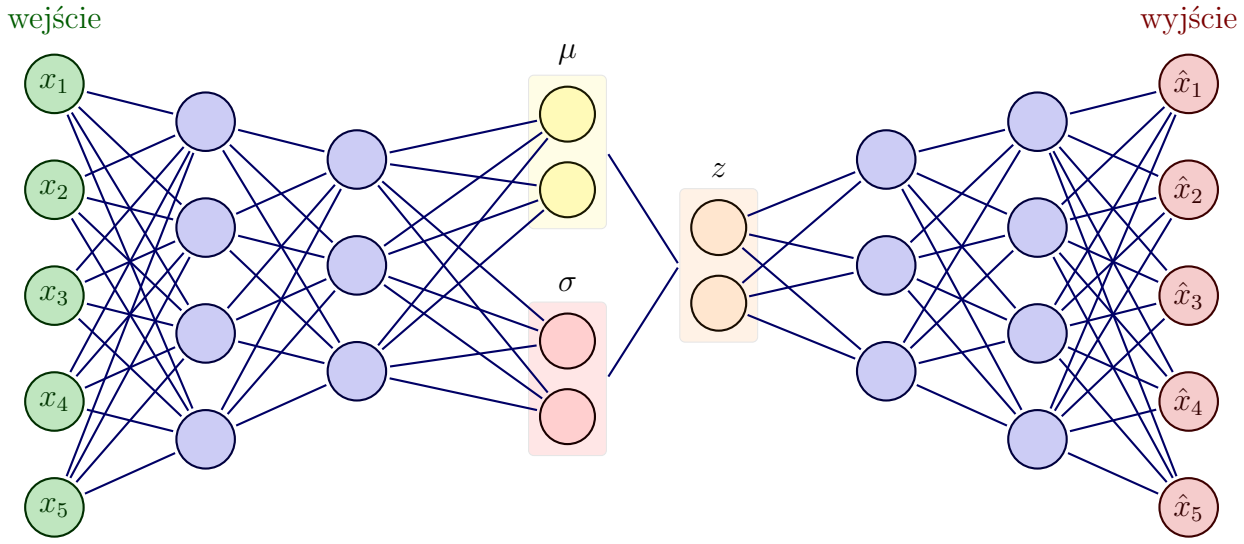
Rysunek 2.1: Schemat budowy wariacyjnego autoenkodera.

Ograniczając enkoder do nauki wyłącznie tej dystrybucji z której próbkujemy losowo zmienne ukryte na których podstawie rekonstruujemy obrazy, zapewniamy sobie 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 dwoma niepołączonymi ze sobą warstwa-



Rysunek 2.2: Wielowymiarowy rozkład zmiennej.



Rysunek 2.3: Sieć neuronowa budująca model VAE.

mi reprezentującymi średnią oraz odchylenie standardowe normalnej dystrybucji, której chcemy 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, reprezentującą długość wektora reprezentującego skompresowane dane.

2.2 Formuła matematyczna

Model VAE mimo swojego podobieństwa do tradycyjnego autoenkodera znacznie różni się w formulacji matematycznej. Największe różnice obserwujemy w zachowaniu enkodera. Kod, który on produkuje, nazywamy wartościami ukrytymi, ponieważ musimy go wywnioskować na podstawie każdej danej. Trenując tradycyjny model nie patrzymy na to, co wiemy o dystrybucji $p(z|x)$, z której pochodzą zmienne. Enkoder wariacyjnego autoenkodera jest, jak już zostało powiedziane, zobowiązany do nauki dystrybucji naszego wyboru. Wyrażenie $p(z|x)$ rozumiemy jako dystrybucję generującą zmienne ukryte na podstawie przedstawionej danej x. Chcemy policzyć ten rozkład. Twierdzenie Bayesa mówi nam że:

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

Aby obliczyć rozkład marginalny $p(x)$ musimy policzyć:

$$p(x) = \int_z p(x|z)p(z)dz$$

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

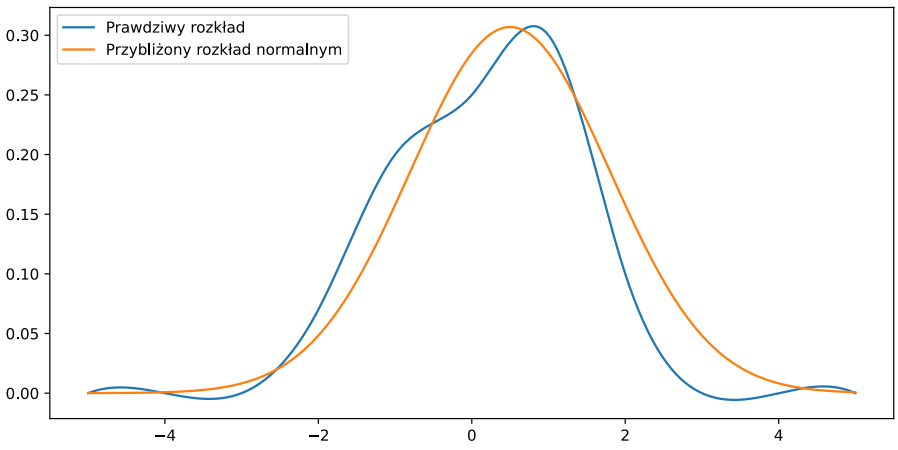
- próbkowanie Monte Carlo łańcuchami Markowa
- wnioskowanie wariacyjne

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.

2.3 Wnioskowanie wariacyjne

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

Wykres 2.3 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 musimy posiadać miarę, która określi nam rozbieżność między dwoma rozkładami prawdopodobieństwa. Miarą tą jest dywergencja Kullbacka-Leiblera. Podstawowymi jej własnościami są:

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

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

$$q^*(z|x) = \text{argmin}_{q(z|x)\in Q}(D_{KL}(q(z|x)\|p(z|x)))$$

gdzie Q to rodzina prostych dystrybucji, na przykład rozkładu Gaussa

2.3.2 Dolna granica dowodów

Zapiszmy że:

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

Policzenie $q(z|x)$ jest proste, ponieważ sami dobieramy sobie jaką dystrybucją jest q. W naszym przypadku jest to rozkład normalny. Mimo to, obliczenia nie są możliwe, ponieważ natrafiamy znowu na problem wyznaczenia $p(z|x)$. Tym razem możemy przepisać:

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

Podstawmy teraz zapisaną inaczej wartość p do wzoru na dywergencję.

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

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

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

Mnożąc $q(z|x)$ przez nawias, otrzymujemy sumę pod całką, co możemy zapisać jako sumę dwóch całek. Następnie wartość $\log p(x)$ możemy wyłączyć przed całkę, ponieważ całkując po z traktujemy ten logarytm jako stałą.

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

Wartość całki, zaznaczonej jako α jest równa 1, dlatego że całkujemy funkcję dystrybucji prawdopodobieństwa po z, więc warunek dla x nie gra dla nas roli. Daje to nam:

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

Na razie nie dokonaliśmy żadnych obliczeń, tylko przepisaliśmy wzór na dywergencję.

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

Wartość $\log p(x)$ nosi nazwę dowodu, ponieważ mówi o prawdopodobieństwie otrzymania obserwacji x przez nasz model. Parametr $-\mathcal{L}$ nazywamy 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 a w naszym przypadku nawet dodatnia.

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

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

$$\operatorname{argmin} D_{KL}(q(z|x)\|p(z|x)) = \operatorname{argmin} \mathcal{L} = \operatorname{argmax} -\mathcal{L} \tag{2.1}$$

2.3.3 Funkcja straty

Wiedząc, z twierdzenia Bayesa, że $p(x,z) = p(x|z)p(z)$ możemy przepisać $\mathcal{L}$ jako:

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

Korzystając kolejny raz z własności logarytmów zapisujemy:

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

Interpretując poszczególne składniki dostajemy:

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

Równanie 2.1 pokazuje, że minimalizacja pierwotnej dywergencji oznacza również minimalizację $\mathcal{L}$.

Pierwsze wyrażenie reprezentuje odległość pomiędzy naszą dystrybucją zastępującą $p(z|x)$, czyli dystrybucją nauczona przez enkoder, a $p(z)$, czyli rozkład zmiennych ukrytych, który jest przez nas wybierany. W naszym przypadku będzie to 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 jesteśmy w stanie zapisać jako rozkłady prawdopodobieństwa. $q(z|x)$ powie nam jakie zmienne ukryte z reprezentują daną x, natomiast $p(x|z)$ generuje nam 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 $p(z) = \mathcal{N}(0, \boldsymbol{I})$. Jeśli weźmiemy wszystkie nasze dane x i przepuścimy je przez enkoder, wiemy, że próbkowane wartości przekazywane następnie do dekodera będą rozłożone normalnie. Rozwiązuje nam to problem tradycyjnego autoenkodera, którego zmienne ukryte są rozrzucone w sposób, który nie jesteśmy w stanie kontrolować. Znając już naszą dystrybucję możemy policzyć dywergencję, pomiędzy dwoma rozkładami normalnymi.

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

Co w naszym przypadku gdzie $\mu_1 = 0$ oraz $\sigma_1 = 1$ uprości się do:

$$\frac{1}{2} \sum_{D}^{i=1} \sigma_i^2 + \mu_i^2 - 2 \log(\sigma_i) - 1 \tag{2.2}$$

gdzie D to długość wektora zmiennych ukrytych. Jest to pierwsza część naszej funkcji straty.

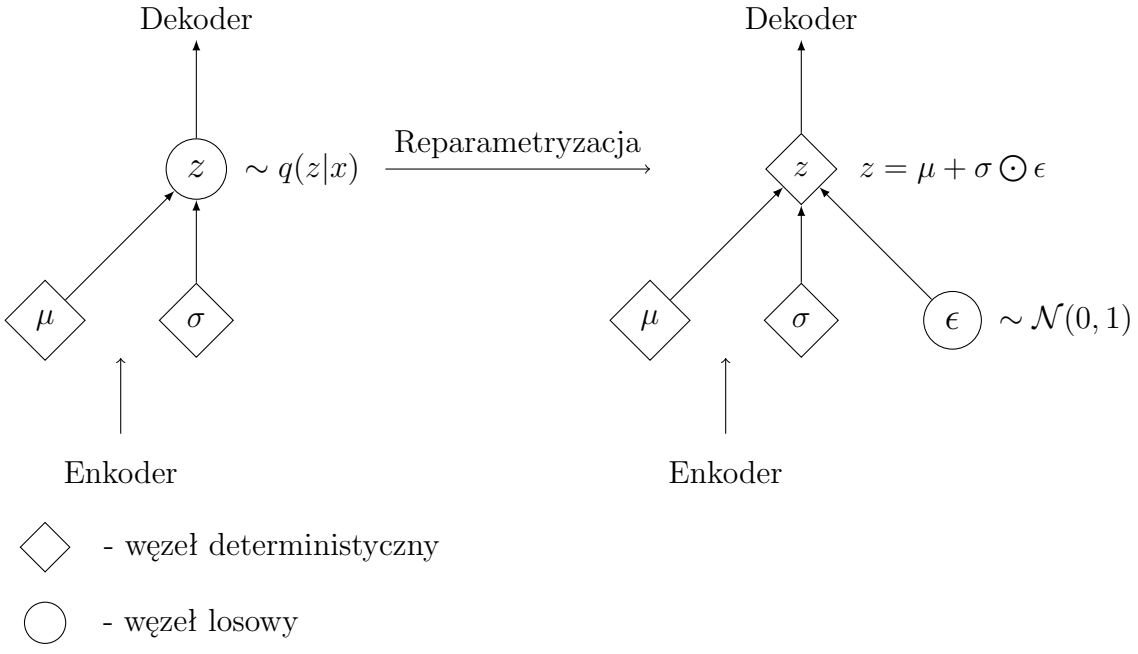
Druga część funkcji straty jest nazywana błędem rekonstrukcji. Możemy zastąpić wartość oczekiwaną wartością jednej próbki, ponieważ do trenowania wag modeli używamy metody gradientu stochastycznego. Aby uniknąć liczenia skompilowanych dystrybucji prawdopodobieństwa $\log p(x|z)$, gdzie poszczególne wymiary x są zależne od siebie, liczymy proste rozkłady na każdym wymiarze osobno. Aby to osiągnąć używamy binarnej entropii krzyżowej. W praktyce często stosujemy również inne funkcje takie jak błąd średnio-kwadratowy, będący bardziej intuicyjnym rozwiązaniem. Możemy tak zrobić, ponieważ wartość prawdopodobieństwa należy do przedziału $[0, 1]$, więc ujemny logarytm z małej liczby (słabego odwzorowania wejścia przez dekoder) będzie ogromna liczą, a z bliskiej jedynki małą. W ten sam sposób zachowuje się średni błąd kwadratowy.

2.3.4 Sztuczka 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 nie możliwe. Operacja próbkowania nie jest różniczkowalna co sprawia, że nie możemy policzyć gradientu po którym będziemy schodzić. Sposobem obejścia tego problemu jest zastosowanie sztuczki (*reparameterization trick*). Próbkowanie z dystrybucji $z \sim \mathcal{N}(\mu, \sigma)$ jesteśmy w stanie zapisać jako:

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

Pozornie nic się nie zmieniło, jednak teraz jesteśmy w stanie poprowadzić gradient przez z, które jest teraz deterministycznie. W poprzednim przypadku było ono losowe wybierane z dystrybucji.

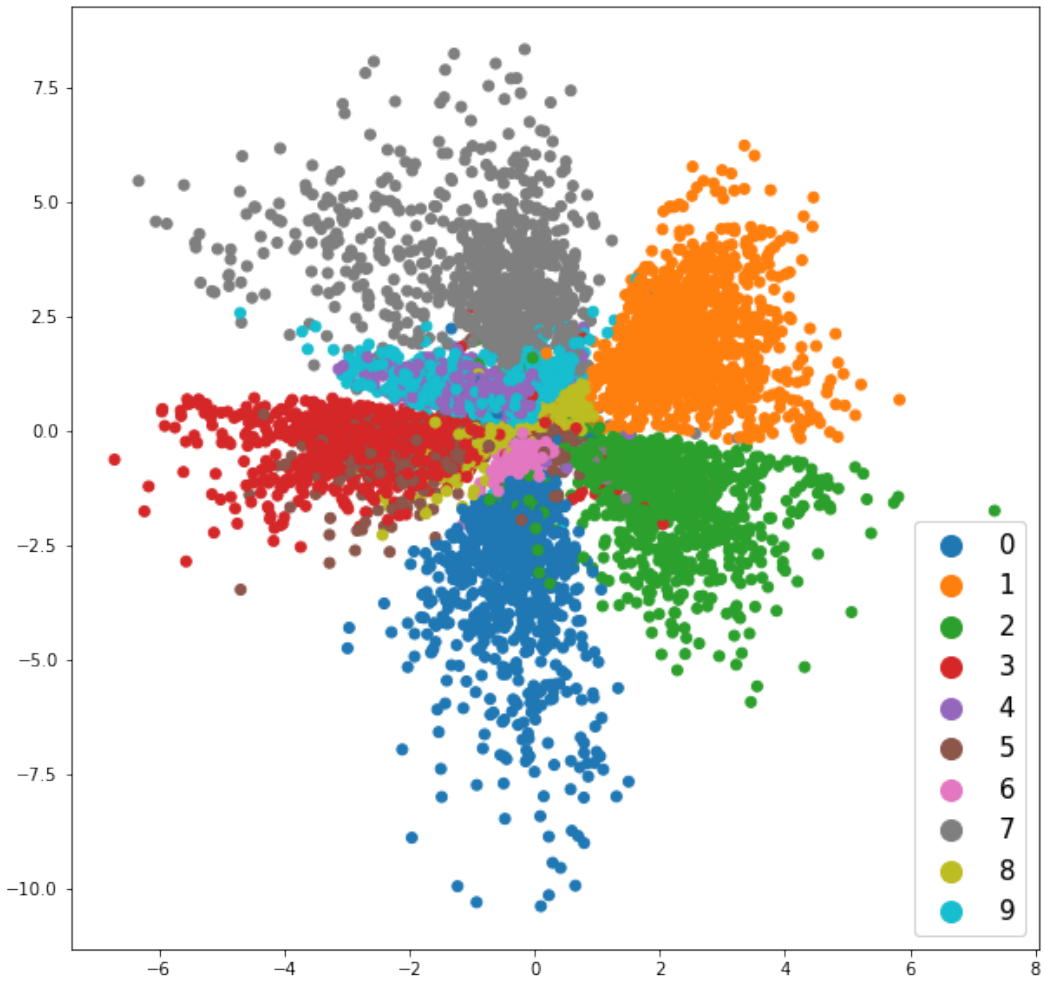


Rysunek 2.5: Graficzna reprezentacja reparametryzacji.

2.4 Zastosowania

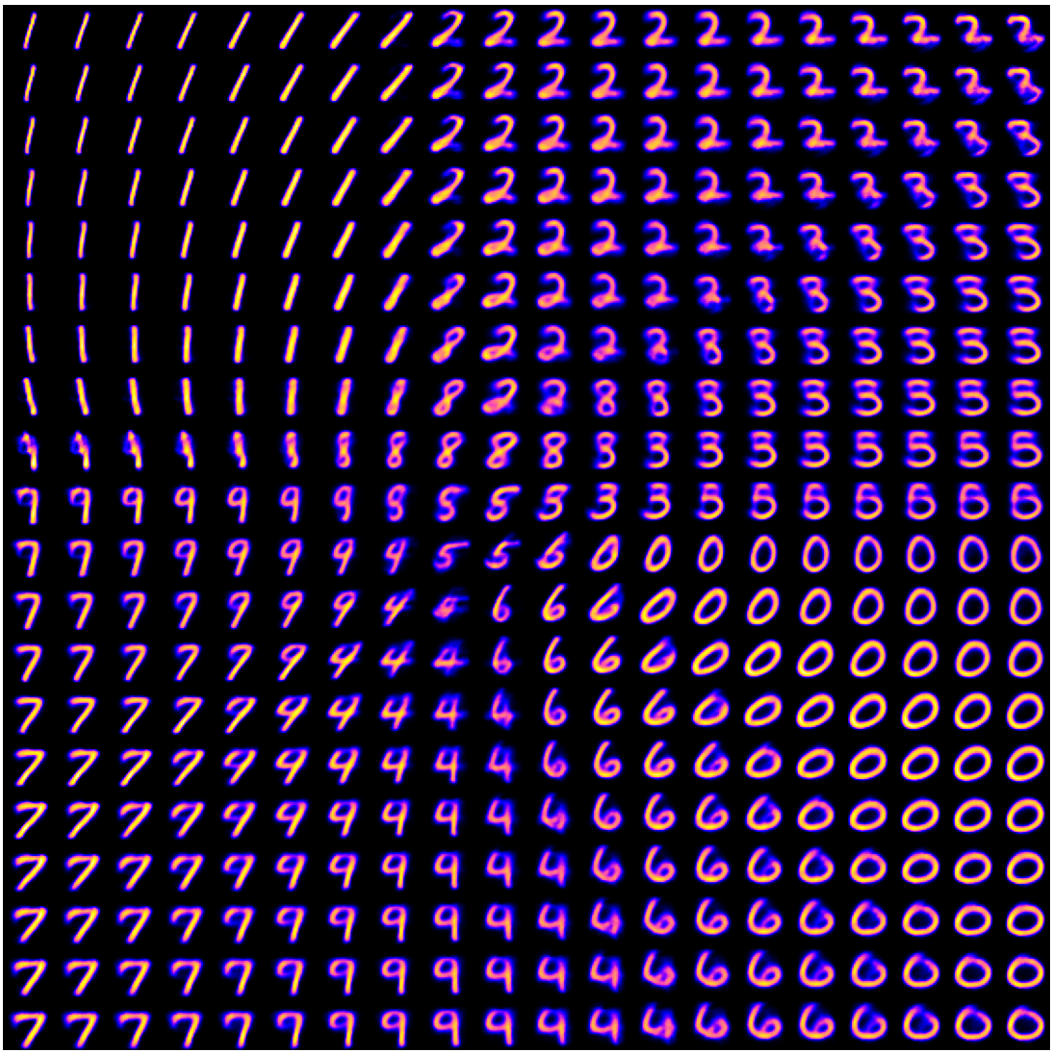
Wariacyjny autoenkoder jako pochodna tradycyjnego modelu, może zostać zastosowany, do takich samych zadań jak, detekcja anomalii, odszumianie oraz kompresja danych. Głównym jednak powodem dla którego chcemy używać modelu VAE jest jego możliwość generowania danych. Mamy już znaną dystrybucję, z której możemy próbkować 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 widzimy, 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.

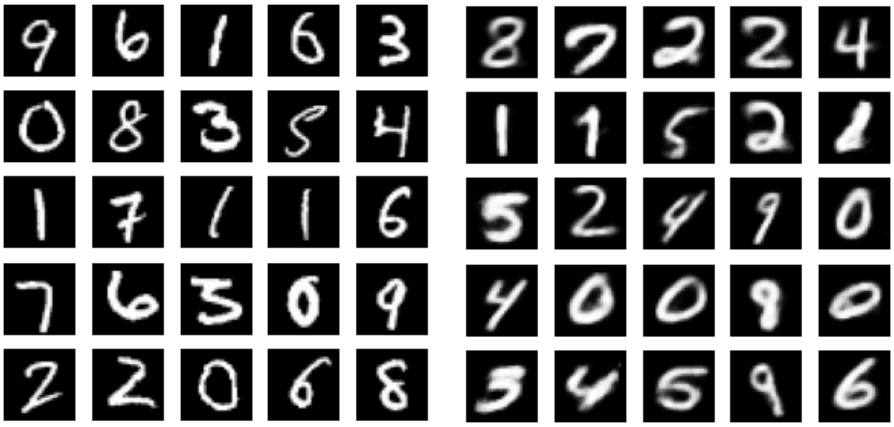


Rysunek 2.6: Przestrzeń zakodowanych zmiennych modelem VAE.

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 potrzebujemy już enkodera. Na wejście dekoder, dostaje losowo wybrane punkty z dystrybucji, którą wybraliśmy sobie 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.

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 wymaga wielu umiejętności, a kod jest mało czytelny, dla wielu osób. 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 będzie NumPy. Jest to wolnych projekt, umożliwiający proste 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

Importujemy potrzebne klasy do implementacji obu modeli.

```
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

```

Następnie wczytujemy zbiór danych MNIST, który podzielony jest na część przeznaczoną do treningu oraz do testów. Implementując modele, nie będziemy korzystać 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, zamieniamy wszystkie próbki na macierze, które przechowują dane w formacie float o długości 32 bitów. Następnie dane skalujemy do przedziału [0, 1] dzieląc każdy piksel każdego obrazka przez 255. Wiemy, że 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. Wyciągamy sobie również wysokość i szerokość obrazów.

```
(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]

```

Autoenkoder

Tworząc model musimy zagwarantować odpowiednią architekturę. W tym przypadku używamy 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 sprawia, że trening zajmuje mniej czasu.[8] Używamy funkcji liniowej w warstwie kodu, aby nie ograniczać wartości jakie poszczególne jego elementu 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 nasze 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. Tworzymy cały model używając 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 musimy stworzyć jego własną warstwę wejściową, a następnie połączyć z nią warstwy, modelu, tak aby korzystał z wytrenowanych parametrów.

```
1  dlugosc_kodu = 2
2  wejscie = Input(shape=(szerokosc,wysokosc))
3  x = Flatten()(wejscie)
4  x = Dense(500, activation='relu')(x)
5  x = Dense(120, activation='relu')(x)
6  kod = Dense(dlugosc_kodu, activation='linear')(x)
7  x = Dense(120, activation='relu')(kod)
8  x = Dense(500, activation='relu')(x)
9  x = Dense(szerokosc * wysokosc, activation='sigmoid')(x)
10 wyjscie = Reshape([szerokosc, wysokosc])(x)
11
12 autoenkoder = Model(wejscie, wyjscie)
13 enkoder = Model(wejscie, kod)
14 dekoder_wejscie = Input(shape=(dlugosc_kodu, ))
15 dec_1 = autoenkoder.layers[5]
16 dec_2 = autoenkoder.layers[6]
17 dec_3 = autoenkoder.layers[7]
18 dec_4 = autoenkoder.layers[8]
19
20 decoder = Model(dekoder_wejscie, dec_4(dec_3(dec_2(dec_1(
       dekoder_wejscie)))))
21
```

Zanim zaczniemy trenować naszą sieć musimy sprecyzować funkcję straty oraz optymalizator. Funkcja użyta w tym przykładzie to błąd średnio-kwadratowy, który jest dobrym wyborem w przypadku obrazów. Optymalizator to algorytm znajdowania wag sieci, dla których funkcja straty jest jak najmniejsza, a co za tym idzie nasz model działa jak najlepiej. Wybraną metodą jest algorytm *adam*, który jest pochodną stochastycznego gradientu, jednak obliczenia okupują mniej pamięci, jest prosty obliczeniowo oraz znajduje optymalne rozwiązanie szybciej niż tradycyjne podejście.[9] Jednym z jej twórców jest Diedrik Kingma, który również jako pierwszy zaproponował model wariacyjnego autoenkodera. Aby nie przetrenować modelu, dodajemy zbiór walidacyjny, który przejmuje

20% danych treningowych. Walidacja służy nam do sprawdzenia na innych danych niż testowe i treningowe, czy model się nie przetrenował. W momencie, kiedy błąd na zbiorze walidacyjnym będzie rosnąć zamiast maleć przez 3 epoki, 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])

```

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

Wariacyjny autoenkoder

Zbiór danych przeznaczony dla wariacyjnego autoenkodera nie różni się w żaden sposób od pokazanego wyżej.

Implementacja modelu VAE jest nieco 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. Warstwy reprezentujące średnią i odchylenie standardowe są połączone z ostatnią warstwą ukrytą. W naszym przypadku, traktujemy jedną warstwę jako logarytm z odchylenia standardowego, ponieważ jego wartość nie może być ujemna. Kiedy chcemy użyć zmiennych wystarczy że dokonamy na nich operacji *exp*. Funkcja *probka* dokonuje próbkowania z dystrybucji na nauczonych parametrach stosując opisaną wcześniej sztuczkę 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 jeszcze nie znamy wielkości batch-a, więc używając metody *shape* będziemy ją dynamicznie zmieniać w zależności od przekazanych parametrów. Dostępna w paczce Keras warstwa *Lambda* pozwala nam wywołać wskazaną funkcję na wartościach neuronów. Warstwę łączymy z obiema wcześniejszymi warstwami na raz. Implementacja dekodera jest identyczna jak dla tradycyjnego autoenkodera. Funkcja strady modelu VAE nie jest możliwa do automatycznego policzenia tak jak zostało to pokazane w AE. Zmienna *z_odkodowane* będzie przechowywać 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)

```

```
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])

```

Aby policzyć wartość funkcji straty tworzymy nową warstwę po ostatniej. Jej jedynym zadaniem jest policzenie straty i nie ma ona możliwych do trenowania parametrów. Pierwsza funkcja oblicza wartość funkcji, a druga pozwala jej policzenie i minimaliza-

cje przez model. Aby porównać obrazy muszą być one w takich samych wymiarach co gwarantujemy przez 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 2.2. Warstwę obliczająca stratę jest połączona z wejściem i wyjściem całego modelu, co umożliwia dostęp do wejścia ze zbioru danych oraz obrazów odkodowanych ze zmiennych ukrytych.

Posiadając warstwę obliczającą funkcje straty możemy stworzyć model. Podczas kompilacji nie podajemy parametru *loss*, ponieważ został on już dodany w naszej niestandardowej warstwie. Z tego samego powodu w funkcji mającej dokonać treningu modelu nie przekazujemy danych, do których jest porównywany wynik autoenkodera.

```
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
```

Spis rysunków

Bibliografia

[1] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[2] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," 2021.

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[4] M. Telgarsky, "Benefits of depth in neural networks," 2016.

[5] R. Eldan and O. Shamir, "The power of depth for feedforward neural networks," 2016.

[6] J. Gramack and A. Gramacki, "Wybrane metody redukcji wymiarowości danych oraz ich wizualizacji," 2008.

[7] E. Plaut, "From principal subspaces to principal components with linear autoencoders," 2018.

[8] A. B. Xavier Glorot and Y. Bengio, "Deep sparse rectifier neural networks," 2011.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.