POLITECHNIKA
LUBELSKA
WYDZIAŁ PODSTAW
TECHNIKI

KIERUNEK: MATEMATYKA

Praca magisterska

Przegląd autoenkoderów stosowanych w nienadzorowanym uczeniu maszynowym

An overview of autoencoders used in unsupervised machine learning

Praca wykonana pod kierunkiem:
dra Dariusza Majerka

Autor:
Alicja Hołowiecka
nr albumu: 89892

Lublin 2022

Spis treści

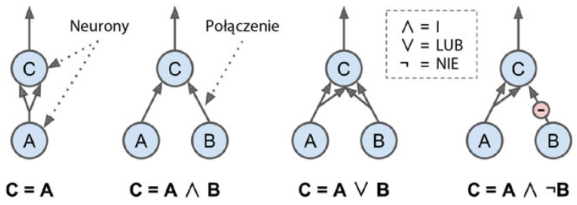
Wstęp

Rozdział 1

Przegląd autoenkoderów

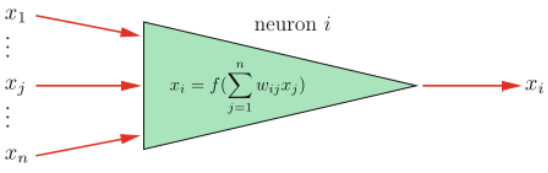
1.1. Sztuczne sieci neuronowe

Sztuczny neuron ma co najmniej jedno binarne wejście i dokładnie jedno binarne wyjście. Wyjście jest uaktywniane, jeżeli jest aktywna określona liczba wejść. Na rysunku 1.1 przedstawione są przykładowe sztuczne sieci neuronowe (SSN lub z angielskiego ANN) wykonujące różne operacje logiczne, przy założeniu, że neuron uaktywni się, gdy przynajmniej dwa wejścia będą aktywne [6].



Rysunek 1.1: Przykładowe sztuczne sieci neuronowe rozwiązujące proste zadania logiczne
Źródło: [6]

Jedną z najprostszych architektur SSN jest **perceptron**, którego podstawą jest sztuczny neuron zwany **progową jednostką logiczną** (ang. *Threshold Logic Unit* - TLU) lub **liniową jednostką progową** (ang. *Linear Threshold Unit* - LTU). Wartościami wejść i wyjść są liczby, a każde połączenie ma przyporządkowaną wagę. Jednostka TLU oblicza ważoną sumę sygnałów wejściowych, a następnie zostaje użyta funkcja skokowa. Schemat takiej jednostki został przedstawiony na rysunku 1.2.



Rysunek 1.2: Struktura sztucznego neuronu, który stosuje funkcję skokową f na ważonej sumie sygnałów wejściowych
Źródło: [5]

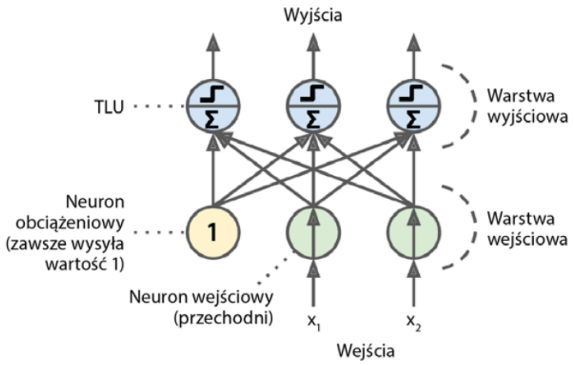
Często używaną funkcją skokową jest **funkcja Heaviside'a**, określona równaniem

$$H(z) = \begin{cases} 0, & \text{jeśli } z < 0 \\ 1, & \text{jeśli } z \geqslant 0 \end{cases}$$

Czasami stosuje się również **funkcję signum**

$$sgn(z) = \begin{cases} -1 & \text{jeśli } z < 0 \\ 0, & \text{jeśli } z = 0 \\ 1, & \text{jeśli } z > 0 \end{cases}$$

Perceptron jest złożony z jednej warstwy jednostek TLU, w której każdy neuron jest połączony ze wszystkimi wejściami. Tego typu warstwa jest nazywana **warstwą gęstą**. Warstwa, do której są dostarczane dane wejściowe, jest nazywana **warstwą wejściową** (ang. *input layer*). Najczęściej do tej warstwy jest wstawiany również **neuron obciążeniowy** (ang. *bias neuron*) $x_0 = 1$, który zawsze wysyła wartość 1. Na rysunku 1.3 znajduje się perceptron z dwoma neuronami wejściowymi i jednym obciążeniowym, a także z trzema neuronami w warstwie wyjściowej.



Rysunek 1.3: Perceptron z trzema neuronami wejściowymi i trzema wyjściami
Źródło: [6]

Obliczanie sygnałów wyjściowych w warstwie gęstej przedstawia się wzorem

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

gdzie $\mathbf{X}$ - macierz cech wejściowych, $\mathbf{W}$ - macierz wag połączeń (oprócz neuronu obciążeniowego), $\mathbf{b}$ - wektor obciążeń zawierający wagi połączeń neuronu obciążeniowego ze wszystkimi innymi neuronami, ϕ - tzw. **funkcja aktywacji**, w przypadku TLU jest to funkcja skokowa.

Algorytm uczący, który służy do trenowania perceptronu, jest silnie inspirowany działaniem neuronu biologicznego. Gdy biologiczny neuron często pobudza inną komórkę nerwową, to połączenia między nimi stają się silniejsze. Reguła ta jest nazywana **regułą Hebba**.

Perceptrony są uczone za pomocą odmiany tej reguły, w której połączenia są wzmacniane, jeśli pomagają zmniejszyć wartość błędu. Dokładniej, w danym momencie perceptron przetwarza jeden przykład uczący i wylicza dla niego predykcję. Na każdy neuron wyjściowy odpowiadający za nieprawidłową prognozę następuje zwiększenie wag połączeń ze wszystkimi wejściami przyczyniającymi się do właściwej prognozy. Aktualizowanie wag przedstawia się następującym wzorem

$$\Delta w_{ij} = \eta(y_j - \hat{y_j})x_i$$

gdzie w_{ij} - waga połączenia między i-tym neuronem wejściowym a j-tym neuronem wyjściowym, x_i - i-ta wartość wejściowa bieżącego przykładu uczącego, $\hat{y_j}$ - wynik j-tego neuronu wyjściowego dla bieżącego przykładu uczącego, y_j - docelowy wynik j-tego neuronu, η - współczynnik uczenia.

Perceptron ma wiele wad związanych z niemożnością rozwiązania pewnych trywialnych problemów (np. zadanie klasyfikacji rozłącznej czyli XOR). Część tych ograniczeń można wyeliminować, stosując architekturę SSN złożoną z wielu warstw perceptronów, czyli **perceptron wielowarstwowy** (ang. *Multi-Layer Perceptron*). Składa się on z jednej warstwy wejściowej (przechodniej), co najmniej jednej warstwy jednostek TLU - tzw. **warstwy ukryte** (ang. *latent layers*) i ostatniej warstwy jednostek TLU - warstwy wyjściowej. Oprócz warstwy wejściowej każda warstwa zawiera neuron obciążający i jest w pełni połączona z następną wartswą. Sieć zawierająca wiele warstw ukrytych nazywamy **głęboką siecią neuronową** (ang. *Deep Neural Network* - DNN).

Do uczenia perceptronów wielowarstwowych wykorzystywany jest algorytm **propagacji wstecznej** (ang. *backpropagation*). Propagacja wsteczna jest właściwie algorytmem gradientu prostego [13]. Można go zapisać jako

$$w_{updated} = w_{old} - \eta \nabla E$$

gdzie E jest funkcją kosztu (funkcją straty) [13]. Proces jest powtarzany do momentu uzyskania zbieżności z rozwiązaniem, a każdy przebieg jest nazywany **epoką** (ang. *epoch*).

Uwaga 1.1. Wagi połączeń wszystkich warstw ukrytych należy koniecznie zainicjować losowo. W przeciwnym przypadku proces uczenia zakończy się niepowodzeniem. Na przykład jeśli wszystkie wagi i obciążenia zostaną zainicjowane wartością 0, to model będzie działał tak, jak gdyby składał się tylko z jednego neuronu. Przy zainicjowaniu wag losowo, symetria zostanie złamana i algorytm propagacji wstecznej będzie w stanie wytrenować zespół zróżnicowanych neuronów [6].

Aby algorytm propagacji wstecznej działał prawidłowo, kluczową zmianą jest zastąpienie funkcji skokowej przez inne **funkcje aktywacji**. Zmiana ta jest konieczna, ponieważ funkcja skokowa zawiera jedynie płaskie segmenty i przez to nie pozwala korzystać z gradientu.

Najczęściej używana jest **funkcja logistyczna (sigmoidalna)**

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Ma ona w każdym punkcie zdefiniowaną pochodną niezerową, dzięki czemu algorytm gradientu prostego może na każdym etapie uzyskać lepsze wyniki. Zbiór wartości tej funkcji wynosi od 0 do 1.

Inną popularną funkcją aktywacji jest **tangens hiperboliczny**

$$\tanh(z) = 2\sigma(2z) - 1$$

Funkcja ta jest ciągła i różniczkowalna, a jej zakres wartości wynosi -1 do 1. Dzięki temu zakresowi wartości wynik każdej warstwy jest wyśrodkowany wobec zera na początku uczenia, co często pomaga w szybszym uzyskaniu zbieżności.

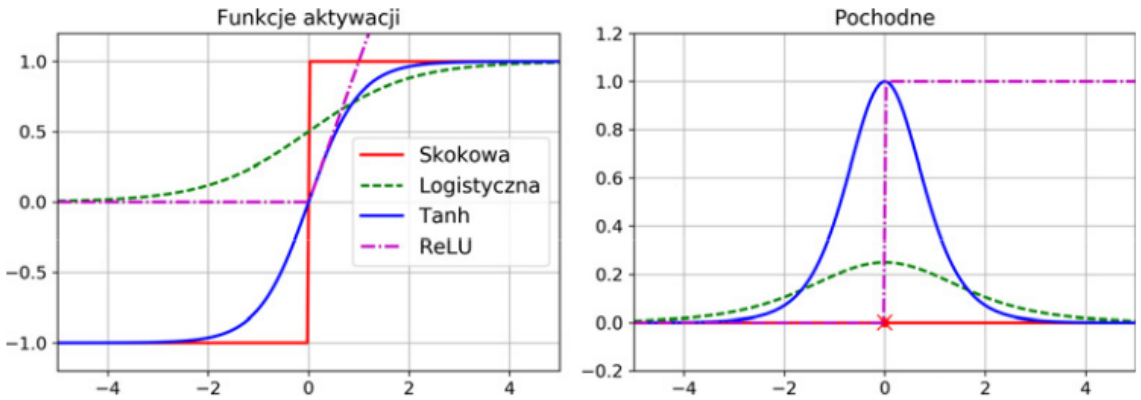
Wśród popularnych funkcji aktywacji należy także wyróżnić **funkcję ReLU** (ang. *Rectified Linear Unit* - prostowana jednostka liniowa) o wzorze

$$ReLU(z) = max(0, z).$$

Jest ona ciągła, ale nieróżniczkowalna w punkcie 0. Jej pochodna dla $z < 0$ wynosi zero. Jej atutem jest szybkość przetwarzania. Nie ma ona maksymalnej wartości wyjściowej. W zadaniach regresji bywa wykorzystywany „wygładzony" wariant funkcji ReLU, czyli funkcja **softplus**:

$$softplus(z) = log(1 + exp(z))$$

Na rysunku 1.4 przedstawiono popularne funkcje aktywacji wraz z ich pochodnymi.



Rysunek 1.4: Przykładowe funkcje aktywacji wraz z pochodnymi

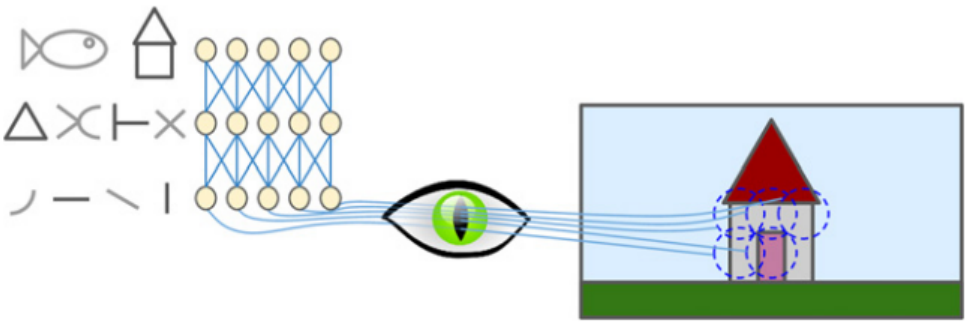
Źródło: [6]

1.2. Sieci splotowe

Splotowe sieci neuronowe (ang. *convolutional neural networks*, CNN) są rodzajem sieci neuronowych służących do przetwarzania danych o znanej topologii siatki. Przykładem takich danych są szeregi czasowe, które można uznać za jednowymiarową siatkę z próbkami w regularnych odstępach czasu, oraz dane graficzne, które można interpretować jako dwuwymiarową siatkę pikseli. Nazwa sieci splotowych pochodzi od wykorzystywanego przez te sieci działania matematycznego nazywanego **splotem** (konwolucją). Można powiedzieć, że sieci splotowe to po prostu sieci neuronowe, które w przynajmniej jednej z warstw zamiast ogólnego mnożenia macierzy wykorzystują splot [7]. Splotowe sieci neuronowe stanowią wynik badań nad korą wzrokową. Od lat 80-tych XX wieku są używane głównie rozpoznawania obrazów, w zagadnieniach takich jak klasyfikacja, detekcja obrazów, transfer stylu. Stanowią podstawę usług takich jak wyszukiwanie obrazu, inteligentne samochody, automatyczne systemy klasyfikowania filmów. Są skuteczne również w innych dziedzinach niż jedynie komputerowe widzenie - takie dziedziny to na przykład rozpoznawanie mowy (*voice recognition*) oraz przetwarzanie języka naturalnego (*Natural Language Processing, NLP*) [6].

1.2.1. Działanie kory wzrokowej

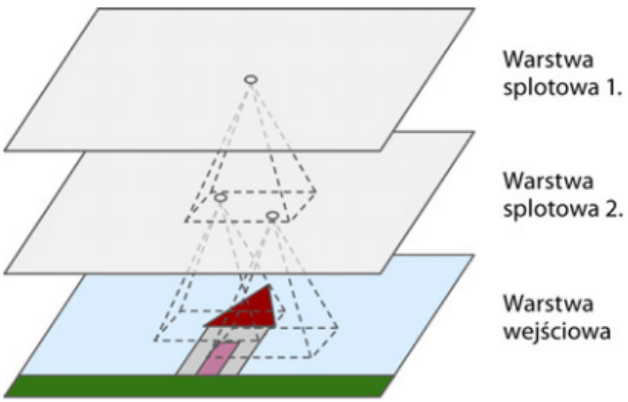
Ponieważ splotowe sieci neuronowe są silnie inspirowane działaniem biologicznej kory wzrokowej, to ten podrozdział zostanie poświęcony wyjaśnieniu jej działania. Na podstawie badań prowadzonych pod koniec lat 50-tych XX wieku [8] [9] wykazano, że neurony biologiczne w korze wzrokowej reagują na określone wzorce w niewielkich obszarach pola wzrokowego, zwanych **polami recepcyjnymi**, a w miarę przepływu sygnału wzrokowego przez kolejne moduły w mózgu neurony rozpoznają coraz bardziej skomplikowane wzorce wykrywane w coraz większych polach recepcyjnych. Wiele neuronów stanowiących korę wzrokową tworzy **lokalne pola recepcyjne**, reagujące jedynie na bodźce wzrokowe mieszczące się w określonym rejonie pola wzrokowego, przy czym lokalne pola recepcyjne poszczególnych neuronów mogą się na siebie nakładać. Takie pola recepcyjne łącznie tworzą całe pole wzrokowe. Dodatkowo badacze zauważyli, iż pewne neurony reagują wyłącznie na obrazy składające się z linii poziomych, lub innych linii ułożonych w konkretny sposób (dwa neurony mogą nawet mieć to samo pole recepcyjne, ale reagować na różne ułożenie linii). Badacze stwierdzili, że niektóre komórki nerwowe mają większe pola recepcyjne i wykrywają bardziej skomplikowane kształty. Takie neurony, odpowiedzialne za rozpoznawanie bardziej skomplikowanych kształtów, znajdują się na wyjściu neuronów reagujących na prostsze bodźce. Działanie neuronów biologicznych w korze wzrokowej, zgodnie z powyższym opisem, zostało zobrazowane na rysunku 1.5.



Rysunek 1.5: Działanie neuronów biologicznych w korze wzrokowej
Źródło: [6]

1.2.2. Warstwy splotowe

Przejdziemy teraz do opisu najistotniejszej części sieci CNN, jaką jest warstwa splotowa (*convolutional layer*). Na rysunku 1.6 widoczny jest przykład warstw splotowych z prostokątnymi polami recepcyjnymi. W pierwszej warstwie splotowej, neurony nie są połączone z każdym pikselem obrazu wejściowego (w przeciwieństwie do opisanych wcześniej warstw gęstych), lecz jedynie z pikselami znajdującymi się w polu recepcyjnym danego neuronu. W kolejnej warstwie każdy neuron łączy się wyłącznie z neuronami z niewielkiego obszaru pierwszej warstwy. Dzięki temu sieć koncentruje się na pewnych cechach w pierwszej warstwie, a w drugiej warstwie może je łączyć w bardziej złożone kształty. Taka hierarchiczna struktura w naturalny sposób występuje na zdjęciach, co przyczyniło się do dużej skuteczności sieci splotowych w rozpoznawaniu obrazu.

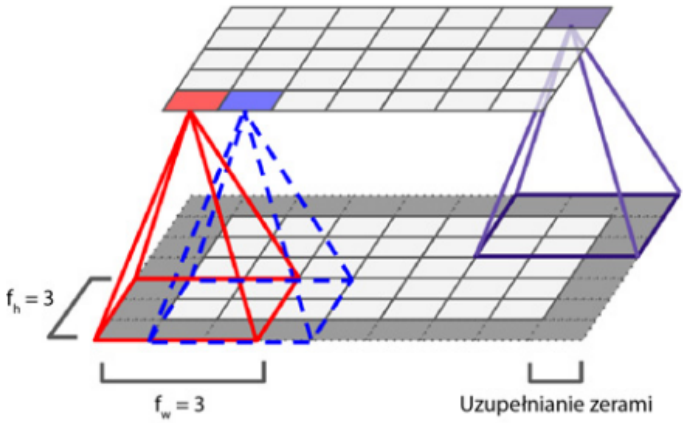


Rysunek 1.6: Warstwy splotowe z prostokątnymi lokalnymi polami recepcyjnymi
Źródło: [6]

Neuron znajdujący się w wierszu i oraz kolumnie j danej warstwy jest połączony z wyjściami neuronów poprzedniej warstwy zlokalizowanymi w rzędach od i do $i + f_h - 1$ i kolumnach od j do $j + f_w - 1$, gdzie f_h i f_w oznaczają, odpowiednio, wysokość i szerokość pola recepcyjnego (rysunek 1.7). W celu uzyskania takich samych wymiarów każdej warstwy

najczęściej są dodawane zera wokół wejść, co zostało pokazane na rysunku 1.7. Proces ten nazywamy **uzupełnianiem zerami** (ang. zero padding).

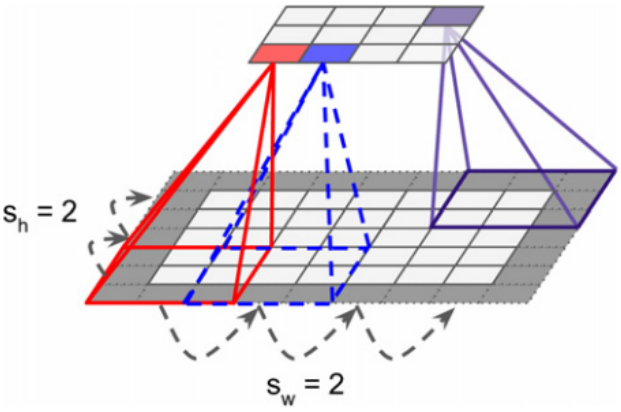
Uwaga 1.2 (Dopełnianie). Dopełnianie (padding) jest ściśle związane z parametrem kroku (który zostanie opisany w kolejnym akapicie). Zapewnia poprawność obliczeń w warstwie konwolucyjnej [10].



Rysunek 1.7: Związek pomiędzy warstwami a uzupełnianiem zerami
Źródło: [6]

Możliwe jest również łączenie bardzo dużej warstwy wejściowej ze znacznie mniejszą kolejną warstwą poprzez rozdzielanie pól recepcyjnych, tak jak zaprezentowano na rysunku 1.8. Rozwiązanie to zmniejsza drastycznie złożoność obliczeniową modelu. Odległość pomiędzy dwoma kolejnymi polami recepcyjnymi nosi nazwę **kroku** (ang. stride). Na widocznym schemacie warstwa wejściowa o wymiarach 5×7 (plus uzupełnianie zerami) łączy się z warstwą o rozmiarze 3×4 za pomocą pól recepcyjnych będących kwadratami 3×3 i kroku o wartości 2 (w omawianym przykładzie krok jest taki sam w obydwu wymiarach, ale nie jest to wcale regułą). Neuron zlokalizowany w rzędzie i oraz kolumnie j górnej warstwy łączy się z wyjściami neuronów dolnej warstwy mieszczącymi się w rzędach od $i \times s_h$ do $i \times s_h + f_h - 1$ i w kolumnach od $j \times s_w$ do $j \times s_w + f_w - 1$, gdzie s_h i s_w definiują wartości kroków odpowiednio w kolumnach i rzędach.

Uwaga 1.3 (Długość kroku). Długość kroku (stride length) - oznacza odległość, o jaką jądro przesuwa się po obrazie. Często stosowaną wielkością jest długość jednego piksela. Często wybieraną długością są również dwa piksele, rzadziej trzy. Dłuższe kroki nie są stosowane, ponieważ jądro może wtedy pomijać obszary obrazu potencjalnie wartościowe dla modelu. Z drugiej strony, im dłuższy krok, tym większa szybkość uczenia się modelu, ponieważ jest mniej obliczeń do wykonania. Trzeba znajdować kompromis między tymi efektami [10].



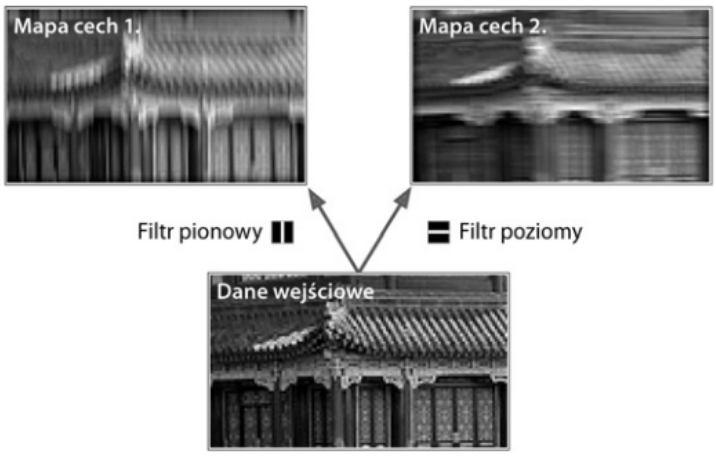
Rysunek 1.8: Warstwa splotowa z krokiem o długości 2
Źródło: [6]

1.2.3. Filtry

Wagi neuronu mogą być przedstawiane jako niewielki obraz o rozmiarze pola recepcyjnego. Na przykład na rysunku 1.9 widzimy dwa możliwe zbiory wag, tak zwane **filtry** (lub **jądra splotowe**; ang. *convolution kernels*). Pierwszy filtr jest symbolizowany jako czarny kwadrat z białą pionową linią przechodzącą przez jego środek (jest to macierz o wymiarach 7×7 wypełniona zerami oprócz środkowej kolumny, która zawiera jedynki); neurony zawierające te wagi będą ignorować wszystkie elementy w polu recepcyjnym oprócz znajdujących się w środkowej pionowej linii (dane wejściowe znajdujące się poza tą linią będą przemnażane przez 0). Drugi filtr wygląda podobnie; różnica polega na tym, że środkowa linia jest ułożona poziomo. Także w tym wypadku będą brane pod uwagę jedynie dane wejściowe znajdujące się w tej linii.

Jeśli wszystkie neurony w danej warstwie będą korzystać z tego samego filtra „pionowego” (i takiego samego członu obciążenia), a my wczytamy do sieci obraz zaprezentowany na dole rysunku 1.9, to uzyskamy obraz widoczny w lewym górnym rogu rysunku. Zauważ, że po zastosowaniu tego filtru pionowe białe linie stają się wyraźniej widoczne, natomiast pozostała część obrazu zostaje rozmazana. Na zasadzie analogii otrzymujemy obraz widoczny w prawym górnym rogu rysunku po zastosowaniu filtru „poziomego”; teraz z kolei białe poziome linie zostają wyostrzone, a reszta obrazu ulega zamazaniu. Zatem warstwa wypełniona neuronami wykorzystującymi ten sam filtr daje nam **mapę cech** (ang. *feature map*), dzięki której możemy dostrzec elementy najbardziej przypominające dany filtr. Sieć CNN w czasie uczenia wyszukuje filtry najbardziej przydatne do danego zadania i uczy się łączyć je w bardziej złożone wzorce.

Uwaga 1.4 (Wielkość jądra). Jądro (zwane również filtrem lub polem receptywnym) typowo ma wysokość i szerokość trzech pikseli. Rozmiar ten okazał się optymalny w szerokim zakresie zastosowań widzenia maszynowego w nowoczesnych sieciach konwolucyjnych. Po-



Rysunek 1.9: Uzyskiwanie dwóch map cech za pomocą dwóch różnych filtrów
Źródło: [6]

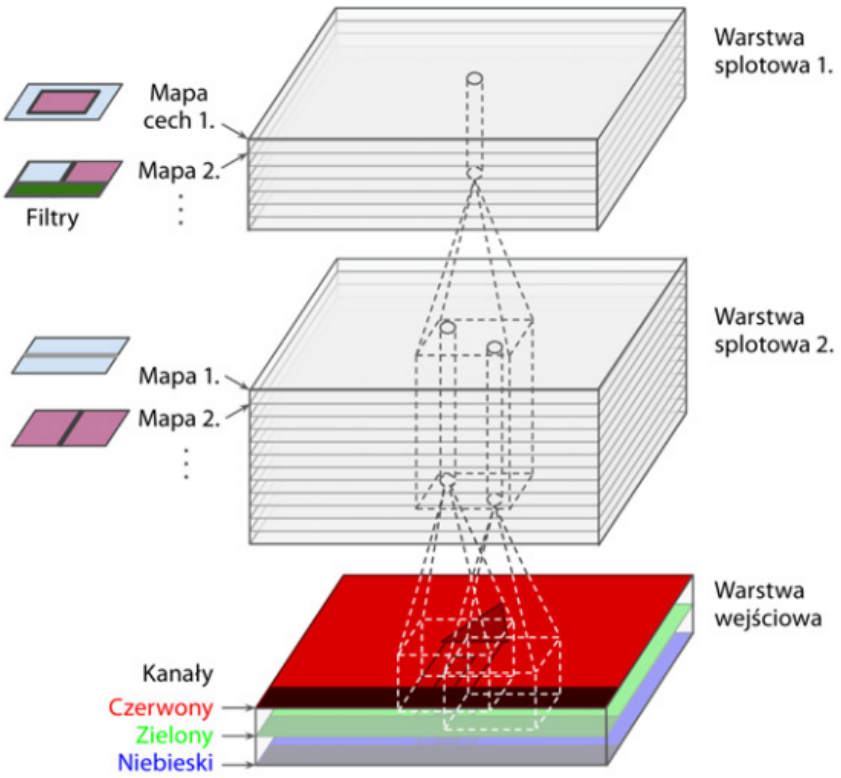
pularna jest również wielkość 5x5 pikseli, a maksymalny stosowany rozmiar to 7x7 pikseli. Jeśli jądro jest zbyt duże w stosunku do obrazu, wtedy w polu receptywnym pojawia się zbyt wiele cech i warstwa konwolucyjna nie jest w stanie się skutecznie uczyć. Jeżeli jądro jest zbyt małe, np. ma wymiary 2x2 piksele, nie jest w stanie dopasować się do żadnej struktury, przez co jest bezużyteczne [10].

1.2.4. Stosy map cech

Do tej pory dla uproszczenia przedstawialiśmy każdą warstwę splotową w postaci cienkiej, dwuwymiarowej warstwy, ale w rzeczywistości składa się ona z kilku map cech o identycznych rozmiarach, dlatego trójwymiarowe odwzorowanie jest bliższe rzeczywistości (rysunek 1.10). W zakresie jednej mapy cech każdy neuron jest przydzielony do jednego piksela, a wszystkie tworzące ją neurony współdzielą te same parametry (wagi i człon obciążenia). Neurony w innych mapach cech mają odmienne wartości parametrów. Pole recepcyjne neuronu nie ulega zmianie, ale „przebiega" przez wszystkie mapy cech poprzednich warstw. Krótko mówiąc, warstwa splotowa równocześnie stosuje różne filtry na wejściach, dzięki czemu jest w stanie wykrywać wiele cech w dowolnym obszarze obrazu.

Co więcej, obrazy wejściowe także składają się z kilku warstw podrzędnych, po jednej na każdy **kanał barw** (ang. *color channel*). Standardowo występują trzy kanały barw — czerwony, zielony i niebieski (ang. red, green, blue — RGB). Obrazy czarno-białe (w odcieniach szarości) zawierają tylko jeden kanał, ale istnieją też takie zdjęcia, które mogą mieć ich znacznie więcej — np. fotografie satelitarne utrwalające dodatkowe częstotliwości fal elektromagnetycznych (takie jak podczerwień).

W szczególności neuron zlokalizowany w rzędzie i oraz kolumnie j mapy cech k w danej warstwie splotowej l jest połączony z neuronami wcześniejszej warstwy $l-1$ umiesz-



Rysunek 1.10: Warstwy splotowe zawierające wiele map cech, a także zdjęcie z trzema kanałami barw

Źródło: [6]

czonymi w rzędach od $i \times s_h$ do $i \times s_h + f_h - 1$ i kolumnach od $j \times s$ w do $j \times s_w + f_w - 1$ we wszystkich mapach cech (warstwy $l-1$). Wszystkie neurony znajdujące się w tym samym rzędzie i oraz kolumnie j, ale w innych mapach cech są połączone z wyjściami dokładnie tych samych neuronów poprzedniej warstwy.

Powyższy opis został podsumowany następującym wzorem, służącym do obliczania wyniku danego neuronu w warstwie splotowej:

$$z_{i,j,k} = b_k \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k}, \quad \text{gdzie} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

W tym równaniu:

— $z_{i,j,k}$ jest wyjściem neuronu znajdującego się w rzędzie i, kolumnie j i mapie cech k warstwy splotowej l;

— jak już zostało wyjaśnione, s_h i s_w to kroki pionowy i poziomy, f_h i f_w są wysokością i szerokością pola recepcyjnego, natomiast $f_{n'}$ oznacza liczbę map cech w poprzedniej warstwie $(l-1)$;

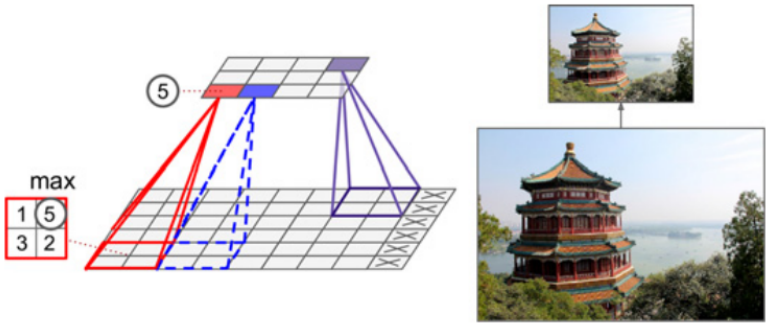
— $x_{i',j',k'}$ jest wyjściem neuronu zlokalizowanego w warstwie $l-1$, rzędzie i', kolumnie j', mapie cech k (lub kanale k', jeżeli poprzednia warstwa była warstwą wejściową);

— b_k to człon obciążenia dla mapy cech k (w warstwie l); można go interpretować jako „pokrętło jasności" mapy cech k;
— $w_{u,v,k',k}$ jest wagą połączenia pomiędzy dowolnym neuronem w mapie cech k warstwy l a jego wejściem mieszczącym się w wierszu u, kolumnie v (względem pola recepcyjnego neuronu) a mapą cech k'.

1.2.5. Warstwa łącząca

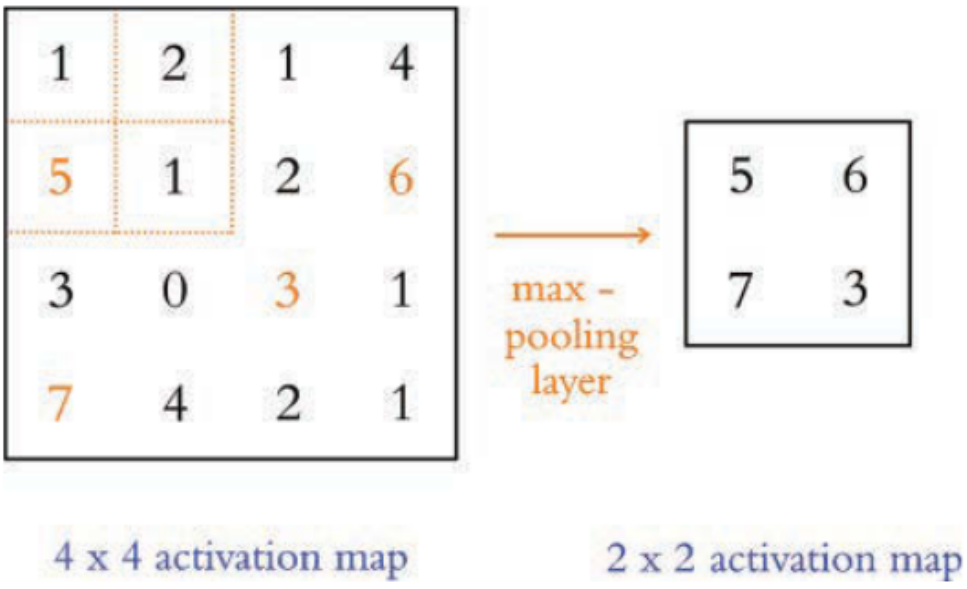
Przejdźmy teraz do drugiego elementu budulcowego sieci CNN — **warstwy łączącej**, zwanej czasem redukującą (ang. pooling layer). Warstwa konwolucyjna może zawierać dowolną liczbę jąder, z których każde generuje mapę aktywacji. Zatem wyjście warstwy konwolucyjnej jest trójwymiarowa tablica aktywacji, której głębokość jest równa liczbie filtrów. Warstwa redukująca zmniejsza przestrzenny wymiar mapy aktywacji, pozostawiając jej głębokość bez zmian [10]. Jej celem jest podpróbkowanie (ang. subsample; tj. zmniejszenie) obrazu wejściowego w celu zredukowania obciążenia obliczeniowego, wykorzystania pamięci i liczby parametrów (a tym samym ograniczenia ryzyka przetrenowania). Podobnie jak w przypadku warstw splotowych, każdy neuron stanowiący część warstwy łączącej łączy się z wyjściami określonej liczby neuronów warstwy poprzedniej, mieszczącej się w obszarze niewielkiego, prostokątnego pola recepcyjnego. Podobnie jak wcześniej, musimy definiować tu rozmiar tego pola, wartość kroku, rodzaj uzupełniania zerami itd. Jednakże warstwa łącząca nie zawiera żadnych wag; jej jedynym zadaniem jest gromadzenie danych wejściowych za pomocą jakiejś funkcji agregacyjnej, np. maksymalizującej lub uśredniającej. Na rysunku 1.11 widzimy najpopularniejszy rodzaj — **maksymalizującą warstwę łączącą** (ang. max pooling layer). W tym przykładzie korzystamy z **jądra łączącego** (ang. pooling kernel) o rozmiarze 2×2, kroku o wartości 2 i z pominięciem uzupełniania zerami. Jedynie maksymalna wartość z każdego jądra zostaje przekazana do następnej warstwy natomiast pozostałe wartości wejściowe zostają odrzucone. Na przykład w lewym dolnym polu recepcyjnym na rysunku 1.11 widzimy wartości wejściowe 1, 5, 3, 2, zatem tylko wartość maksymalna, czyli 5, zostanie przekazana do następnej warstwy. Z powodu kroku równego 2 obraz wyjściowy ma szerokość i wysokość o połowę mniejsze w porównaniu do obrazu wejściowego (zaokrąglamy tu w dół, ponieważ nie korzystamy z uzupełniania zerami).

Uwaga 1.5 (Parametry warstwy redukującej). Filtr w warstwie redukującej ma zazwyczaj wymiary 2x2 piksele, a krok ma długość dwóch pikseli. W takim wypadku filtr w każdej pozycji przetwarza cztery wartości aktywacji, wybiera największą i w efekcie czterokrotnie redukuje liczbę aktywacji [10].



Rysunek 1.11: Maksymalizująca warstwa łącząca (jądro łączące: 2×2, krok: 2, brak uzupełniania zerami)
Źródło: [6]

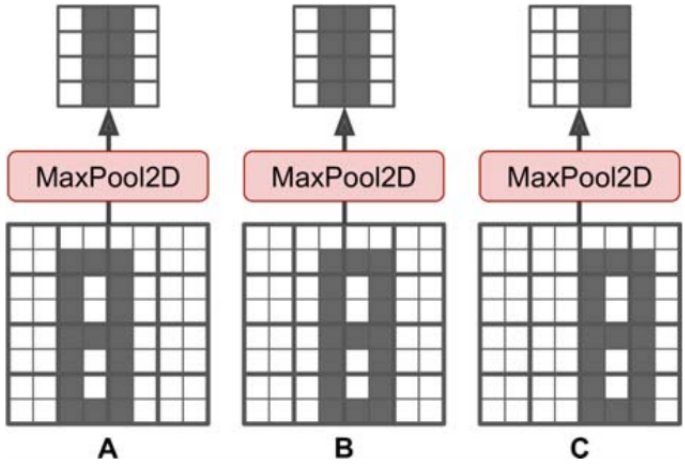
Na rysunku 1.12 widoczne jest działanie maksymalizującej warstwy redukującej na mapie aktywacji o wymiarze 4x4. Filtr przesuwa się nad danymi wejściowymi od lewej do prawej, od góry do dołu, tak jak w warstwie konwolucyjnej, i w każdej zajmowanej pozycji przeprowadza operację redukcji danych. W tym przykładzie filtr i krok mają rozmiar 2x2. Skutkuje to otrzymaniem mapy cztery razy mniejszej niż oryginalna.



Rysunek 1.12: Warstwa *max-pooling* z filtrem i krokiem rozmiaru 2x2, zastosowana do mapy aktywacji o rozmiarze 4x4 (widoczna po lewej stronie) skutkuje uzyskaniem mapy czterokrotnie mniejszej niż oryginalna
Źródło: [10]

Oprócz ograniczania liczby obliczeń, zużycia pamięci i liczby parametrów maksymalizująca warstwa łącząca wprowadza także pewien stopień **niezmienniczości** w stosunku do

drobnych przesunięć, co widać na rysunku 1.13. Zakładamy tu, że piksele jasne mają mniejszą wartość od pikseli ciemnych, trzy obrazy (A, B i C) przechodzą przez maksymalizującą warstwę łączącą o jądrze 2×2 i kroku równym 2. Obrazy B i C wyglądają tak samo jak obraz A, ale są przesunięte o, odpowiednio, jeden i dwa piksele w prawo. Jak widać, rezultaty wygenerowane w maksymalizującej warstwie łączącej z obrazów A i B są identyczne. Na tym polega **niezmienniczość przesunięć** (ang. translation invariance). W przypadku obrazu C wynik jest odmienny: jest on przesunięty o jeden piksel w prawo (nadal jednak pozostaje niezmieniony w mniej więcej 75%). Poprzez wstawianie maksymalizującej warstwy łączącej co kilka warstw sieci CNN możliwe jest uzyskanie ograniczonej niezmienniczości przesunięć w większej skali. Ponadto warstwa ta zapewnia niewielki stopień niezmienniczości rotacyjnej i drobną niezmienniczość skalowania. Tego typu niezmienniczość (mimo że jest ograniczona) przydaje się wszędzie tam, gdzie prognozy nie powinny być zależne od tych szczegółów, na przykład w zadaniach klasyfikacji.



Rysunek 1.13: Niezmienniczość związana z drobnymi przesunięciami
Źródło: [6]

Jednak maksymalizująca warstwa łącząca jest niepozbawiona również wad. Przede wszystkim jest ona bardzo niszczycielska: nawet w przypadku niewielkiego jądra o rozmiarze 2×2 i kroku o wartości 2 dane wyjściowe będą dwukrotnie mniejsze w każdym kierunku (zatem obszar obrazu będzie zmniejszony czterokrotnie), co oznacza porzucenie 75% wartości wejściowych. Z kolei w pewnych zastosowaniach niezmienniczość jest niepożądana, na przykład w segmentacji semantycznej (zadaniu klasyfikowania każdego piksela obrazu zgodnie z jego przynależnością do danego obiektu): jest oczywiste, że jeżeli obraz wejściowy zostanie przesunięty o jeden piksel w prawo, to wynik również powinien być przesunięty w taki sam sposób. Wówczas celem staje się **ekwiwariancja** (ang. equivariance), a nie niezmienniczość: mała zmiana w sygnale wejściowym powinna prowadzić do powiązanej z nią niewielkiej zmiany w sygnale wyjściowym.

Podobnie do maksymalizującej warstwy łączącej jest zdefiniowana **uśredniająca warstwa łącząca** (average pooling layer), która zamiast maksimum używa średniej. Jest ona rzadziej wybierana w zastosowaniach niż warstwa maksymalizująca, z powodu słabszej wydajności. Obliczanie średniej zazwyczaj powoduje mniejszą utratę informacji niż obliczanie maksimum, ale za to warstwa maksymalizująca zachowuje wyłącznie najistotniejsze cechy i ignoruje te mniej ważne, dlatego kolejne warstwy otrzymują coraz czystszy sygnał.

Ostatnim rodzajem warstwy łączącej często spotykanym we współczesnych architekturach jest **globalna uśredniająca warstwa łącząca** (ang. global average pooling layer). Mechanizm jej działania jest całkiem odmienny: oblicza ona jedynie średnią każdej mapy cech (przypomina to działanie uśredniającej warstwy łączącej, w której jądro ma takie same wymiary przestrzenne jak dane wejściowe). Oznacza to, że generuje ona na wyjściu pojedynczą wartość na każdą mapę cech i na każdy przykład. Jest to oczywiście rozwiązanie skrajnie destrukcyjne (większość informacji zawartych w mapie cech zostaje utraconych), ale, jak przekonasz się w dalszej części rozdziału, bywa przydatne na wyjściu modelu.

1.3. Czym jest autoenkoder

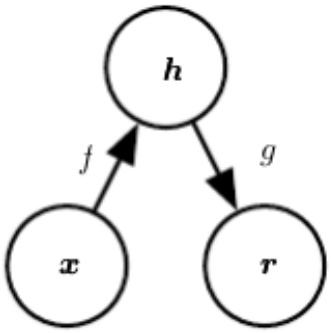
Autoenkoder (czasem także nazywany autokoderem, z ang. *autoencoder*, *auto-encoder*) jest specjalnym typem sieci neuronowej, która jest przeznaczona głównie do kodowania danych wejściowych do skompresowanej i znaczącej reprezentacji, a następnie dekodowania ich z powrotem w taki sposób, aby zrekonstruowane dane były jak najbardziej podobne do oryginalnych [2]. Autoenkodery uczą się gęstych reprezentacji danych, tzw. reprezentacji ukrytych (ang. *latent representations*) lub kodowań (ang. *codings*) bez jakiejkolwiek formy nadzorowania (tzn. zbiór danych nie zawiera etykiet). Wyjściowe kodowania zazwyczaj mają mniejszą wymiarowość od danych wejściowych, dzięki czemu autoenkodery mogą z powodzeniem służyć do redukcji wymiarowości. Mają też zastosowanie w **modelach generatywnych** (ang. *generative models*), które potrafią losowo generować nowe dane przypominające zbiór uczący. Jeszcze lepszej jakości dane można uzyskać przy użyciu **generatywnych sieci przeciwstawnych**, czyli GAN (ang. *Generative Adversial Networks*). Sieci GAN są często stosowane w zadaniach zwiększania rozdzielczości obrazu, koloryzowania, rozbudowanego edytowania zdjęć, przekształcania prostych szkiców w realistyczne obrazy, dogenerowywania danych służących do uczenia innych modeli, generowania innych typów danych np. tekstowych, dźwiękowych, itd. [6].

(Chapter 14 Autoencoders [7])

An autoencoder is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer h that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function h = f (x) and a

decoder that produces a reconstruction r = g(h). This architecture is presented in figure 14.1 . If an autoencoder succeeds in simply learning to set g(f (x)) = x everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

Autoenkoder to sieć neuronowa szkolona po to, aby próbować kopiować swoje wejście do wyjścia. Wewnątrz zawiera ukrytą warstwę h, która opisuje kod używany do reprezentowania wejścia. Sieć można postrzegać jako składającą się z dwóch części: kodującej funkcji $h(x)$ i dekodera, który tworzy rekonstrukcję $r = g(h)$.



Rysunek 1.14: The general structure of an autoencoder, mapping an input x to an output (called reconstruction) r through an internal representation or code h. The autoencoder has two components: the encoder f (mapping x to h) and the decoder g (mapping h to r).
Źródło: [7]

Jeśli autoenkoderowi uda się po prostu nauczyć ustawiać wszędzie $g(f(x)) = x$, to wtedy nie jest zbytnio przydatny. Dlatego autoenkodery projektuje się tak, aby nie potrafiły doskonale kopiować. Zwykle nakłada się ograniczenia, aby mogły kopiować jedynie w przybliżeniu i jedynie wejście, które przypomina dane treningowe. Ponieważ model musi ustanawiać priorytety, aby ustalać, jakie aspekty wejścia powinny być kopiowane, często poznaje przydatne właściwości danych.

We współczesnych zastosowaniach oprócz deterministycznych funkcji do stochastycznego odwzorowania $p_{encoder}(h|x)$ i $p_{decoder}(x|h)$ stosuje się uogólnioną koncepcję kodera i dekodera.

Tradycyjnie autoenkodery były używane do redukcji wymiarów lub poznawania cech. Ostatnio teoretyczne powiązania z modelami zmiennych utajonych sprawiły, że autoenkodery znalazły się na pierwszym planie w modelowaniu generatywnym.

Autoenkodery można wyobrazić sobie jako specjalny przypadek sieci jednokierunkowych i można je szkolić, używając wszystkich tych samych technik, zwykle minipakietowego spadku gradientu po gradientach obliczonych przez propagację wstecz.

W przeciwieństwie to ogólnych sieci jednokierunkowych, autoenkodery można szkolić za pomocą recyrkulacji, czyli algorytmu uczącego się na bazie porównywania aktywacji sieci na oryginalnym wejściu z aktywacjami na zrekonstruowanym wejściu.

1.4. Rodzaje autoenkoderów

Rodzaje autoenkoderów:
- niedopełniony (undercomplete)
- regularyzowany (regularized, overcomplete)
- stosowy (stacked) lub inaczej głęboki (deep)
- splotowy (convolutional)
- rekurencyjny (recurrent)
- odszumiający (stacked denoising)
- rzadki (sparse)
- wariancyjny (variational)

1.4.1. Autoenkodery niedopełnione

[7]

Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in h taking on useful properties. One way to obtain useful features from the autoencoder is to constrain h to have a smaller dimension than x. An autoencoder whose code dimension is less than the input dimension is called undercomplete. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(x, g(f(x)))$$

where L is a loss function penalizing g(f(x)) for being dissimilar from x, such as the mean squared error. When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA.

Kopiowanie wejścia do wyjścia może wydawać się bezużyteczne, ale wyjście dekodera nas nie interesuje. Mamy za to nadzieję, że skutkiem przeszkolenia autoenkodera do kopiowania będzie kod h, mający przydatne właściwości.

Jednym ze sposobów, aby uzyskać przydatne cechy z autoenkodera, jest ograniczenie h do mniejszego wymiaru niż x. Autoenkoder, w którym wymiar kodu jest mniejszy niż wymiar wejściowy, jest nazywany niekompletnym (niedopełnionym).

Poznawanie niekompletnych reprezentacji zmusza autoenkoder do przechwycenia najistotniejszych cech danych szkoleniowych.

Proces poznawania jest opisywany po prostu jako minimalizowanie funkcji straty

$$L(x, g(f(x)))$$

gdzie L jest funkcją straty karzącą $g(f(x))$ za niepodobieństwo do x jak np. błąd średniokwadratowy.

Gdy dekoder jest liniowy, a L to błąd średniokwadratowy, niekompletny autoenkoder uczy się obejmować tą samą podprzestrzeń co PCA. W tym przypadku poznanie zasadniczej podprzestrzeni danych szkoleniowych przez szkolony do kopiowania autoenkoder jest efektem ubocznym. Autoenkodery z nieliniowymi funkcjami kodowania f i nieliniowymi funkcjami dekodowania g mogą więc poznawać potężniejsze, nieliniowe uogólnienie PCA. Niestety, jeśli koder i dekoder będą mieć zbyt dużą pojemność, autoenkoder może nauczyć się wykonywać kopiowanie bez wyodrębniania pożytecznych informacji o rozkładzie danych. Teoretycznie można sobie wyobrazić, ze autoenkoder z jednowymiarowym kodem, ale bardzo potężnym nieliniowym koderem, może nauczyć się reprezentować każdy przykład szkoleniowy $x^{(i)}$ za pomocą kodu i. Dekoder mógłby nauczyć się odwzorowywać te całkowitoliczbowe indeksy z powrotem na wartości konkretnych przykładów szkoleniowych. Ten konkretny przykład nie występuje w praktyce, ale pokazuje wyraźnie, że autoenkoder przeszkolony do wykonywania kopiowania może zawieść, jeśli chodzi o poznanie czegoś przydatnego na temat zbioru danych, jeśli pozwoli się, aby miał za dużą pojemność.

1.4.2. Autoenkodery z regularyzacją

[7]

W przypadku, gdy wymiar wyjścia jest większy (autoenkodery nadkompletne) lub równy niż wymiar wejścia, nawet liniowy koder i dekoder mogą nauczyć się kopiować wejście do wyjścia bez uczenia się niczego przydatnego na temat rozkładu danych.

Ideałem byłaby możliwość udanego szkolenia dowolnej architektury autoenkodera przy wyborze wymiaru kodu oraz pojemności kodera i dekodera na podstawie złożoności rozkładu modelowanego. Można to zrobić, stosując regularyzację. Zamiast ograniczać pojemność modelu przez zachowywanie płytkości kodera i dekodera oraz małego rozmiaru kodu, autoenkodery z regularyzacją używają funkcji straty, dzięki której model może posiadać inne właściwości oprócz możliwości kopiowania swojego wejścia do wyjścia. Do tych właściwości należą:

— rzadkość reprezentacji
— mały rozmiar pochodnej reprezentacji
— odporność na szum lub brakujące dane wejściowe

Autoenkoder z regularyzacją może być nieliniowy i nadkompletny, a mimo to nauczyć się czegoś wartościowego o rozkładzie danych, nawet jeśli pojemność modelu jest na tyle duża, aby poznać trywialną funkcję tożsamościową.

Rzadkie autoenkodery

To autoenkoder, którego kryterium szkolenia obejmuje karę rzadkości $\Omega(h)$ na warstwie kodu h oprócz błędu rekonstrukcji

$$L(x, g(f(x))) + \Omega(h)$$

gdzie $g(h)$ to wyjście dekodera, a zwykle mamy $h = f(x)$, czyli wyjście kodera.

Rzadkie autoenkodery są zwykle używane do tego, aby uczyć się cech do innego zadania, takiego jak klasyfikacja. Autoenkoder, który dzięki regularyzacji jest rzadki, musi reagować na unikatowe statystyczne cechy zbioru danych, na którym został wyszkolony, a nie tylko działać jak funkcja tożsamościowa.

Autoenkodery z odszumianiem

Zamiast dodawać karę Ω do funkcji kosztów, możemy uzyskać autoenkoder, który uczy się czegoś przydatnego, zmieniając składnik błędu rekonstrukcji w funkcji kosztów. Tradycyjne autoenkodery minimalizują jakąś funkcję

$$L(x, g(f(x)))$$

gdzie L to funkcja straty karząca $g(f(x))$ za niepodobieństwo do x, jak np. norma L^2 ich różnicy. Sprzyja to temu, aby $g \circ f$ uczyła się być jedynie funkcją tożsamościową, jeśli ma do tego odpowiednią pojemność.

Autoenkoder z odszumianiem zamiast tego minimalizuje

$$L(x, g(f(\tilde{x})))$$

gdzie $\tilde{x}$ to kopia x, która została zniekształcona przez jakiegoś rodzaju postać szumu. Autoenkodery mają tym samym za zadanie odwrócić to zniekształcenie, a nie po prostu przekopiować swoje wejście.

Regularyzacja poprzez karanie pochodnych

kolejną strategią regularyzacji autoenkodera jest użycie kary Ω tak, jak w rzadkich autoenkoderach

$$L(x, g(f(x))) + \Omega(h, x)$$

ale z inną postacią Ω:

$$\Omega(h,x) = \lambda \sum_i ||\nabla_x h_i||^2$$

Zmusza to model do poznania funkcji, która nie zmienia się za bardzo przy niewielkich zmianach x. Ponieważ kara ta jest stosowana tylko w przykładach szkoleniowych, zmusza autoenkoder do poznawania cech, które przechwytują informacje o rozkładzie szkoleniowym.

Autoenkoder z taką regularyzacją jest nazywany kurczliwym.

1.4.3. Autoenkodery stosowe

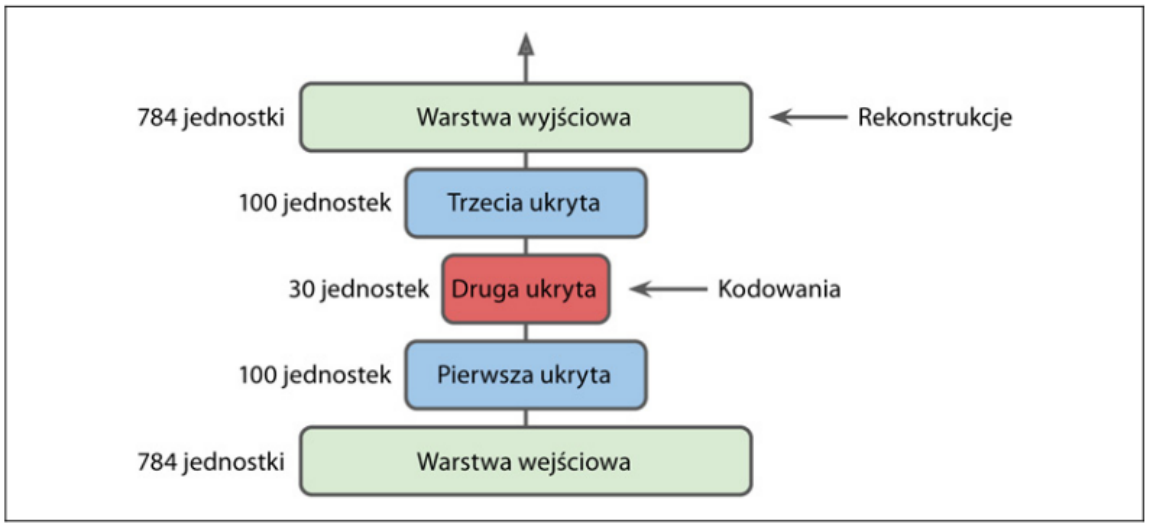
(kopiowane z Gerona:)

Autokodery, podobnie jak w przypadku innych rodzajów sieci neuronowych, również mogą mieć wiele warstw ukrytych. W takiej sytuacji są one nazywane autokoderami stosowymi (ang. stacked autoencoders) lub głębokimi (ang. deep autoencoders). Kolejne warstwy ukryte pozwalają autokode- rowi uczyć się bardziej skomplikowanych kodowań. Jednak musimy uważać, aby nie stworzyć zbyt potężnego modelu. Wyobraź sobie tak wydajny autokoder, że nauczyłby się rzutować każdy przykład wejściowy do postaci dowolnej pojedynczej liczby (a dekoder uczyłby się odwrotnego rzuto- wania). Oczywiście taki autokoder potrafiłby doskonale rekonstruować dane uczące, ale w trakcie tego procesu nie nauczy się żadnej przydatnej reprezentacji danych (i raczej nie będzie w stanie dobrze generalizować przewidywań na nowe próbki). Architektura autokodera stosowego najczęściej jest symetryczna względem centralnej warstwy ukrytej (kodowania). Najprościej mówiąc, przypomina ona kanapkę. Na przykład autokoder klasyfikujący obrazy MNIST (zbiór ten został opisany w rozdziale 3.) może zawierać 784 wejścia, po których nastę- puje stuneuronowa warstwa ukryta, następnie środkowa warstwa ukryta zawierająca 30 jednostek, a po niej kolejna stuneuronowa warstwa ukryta przechodząca w warstwę wyjściową mającą 784 wyj- ścia.

1.4.4. Autoenkoder splotowy

[6]

Koderem jest tradycyjna sieć splotowa składająca się z warstw splotowych i łączących. Zazwyczaj zmniejsza ona wymiarowość danych wejściowych (wysokość i szerokość obrazu), a jednocześnie zwiększa ich głębokość (liczbę map cech). Dekoder musi przeprowadzać odwrotną operację (zwiększyć rozdzielczość obrazu i zredukować głębokość do pierwotnej liczby wymiarów), dlatego możemy w tym celu wykorzystać transponowane warstwy splotowe (ewentualnie możesz łączyć warstwy ekspansji z warstwami splotowymi).



Rysunek 1.15: Stosowy
Źródło: [6]

1.4.5. Autoenkoder rekurencyjny

[6]

Jeżeli chcesz tworzyć autokodery przeznaczone do przetwarzania sekwencji, takich jak szeregi czasowe czy teksty (np. nienadzorowanego uczenia wstępnego lub redukowania wymiarowości), to reku- rencyjne sieci neuronowe (zob. rozdział 15.) mogą nadawać się do tego zadania lepiej niż sieci gęste. Budowanie autokodera rekurencyjnego (ang. recurrent autoencoder) jest proste: koder stanowi zazwyczaj sieć sekwencyjno-wektorową, która kompresuje sekwencję wejściową do postaci poje- dynczego wektora. Dekoder to sieć wektorowo-sekwencyjna przeprowadzająca odwrotną operację

```
recurrent_encoder = keras.models.Sequential([
keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
keras.layers.RepeatVector(28, input_shape=[30]),
keras.layers.LSTM(100, return_sequences=True),
keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

Taki autokoder rekurencyjny może przetwarzać sekwencje o dowolnej długości, które w każdym takcie zawierają 28 wymiarów. Jest to dla nas dogodne, ponieważ oznacza, że możemy traktować obrazy z zestawu Fashion MNIST tak, jakby każdy obraz stanowił sekwencję

rzędów: w każdym takcie sieć rekurencyjna będzie przetwarzać pojedynczy, 28-elementowy rząd pikseli. Oczywiście autokoder rekurencyjny może być stosowany wobec dowolnego rodzaju sekwencji. Zwróć uwagę,że jeśli jako pierwszą warstwę dekodera wstawimy RepeatVector , to musimy sprawić, żeby w każdym takcie do dekodera był dostarczany wektor wejściowy.

Do tej pory zmuszaliśmy autokoder do rozpoznawania interesujących cech, ograniczając rozmiar warstwy kodowania, przez co był on niedopełniony. W rzeczywistości istnieje wiele rodzajów ograniczeń, które możemy wykorzystać, w tym również umożliwiające stosowanie warstwy kodowa- nia o takim samym rozmiarze jak wejściowa, a nawet jeszcze większej, co pozwala uzyskać autokoder przepełniony (ang. overcomplete autoencoder). Sprawdźmy teraz te inne rozwiązania.

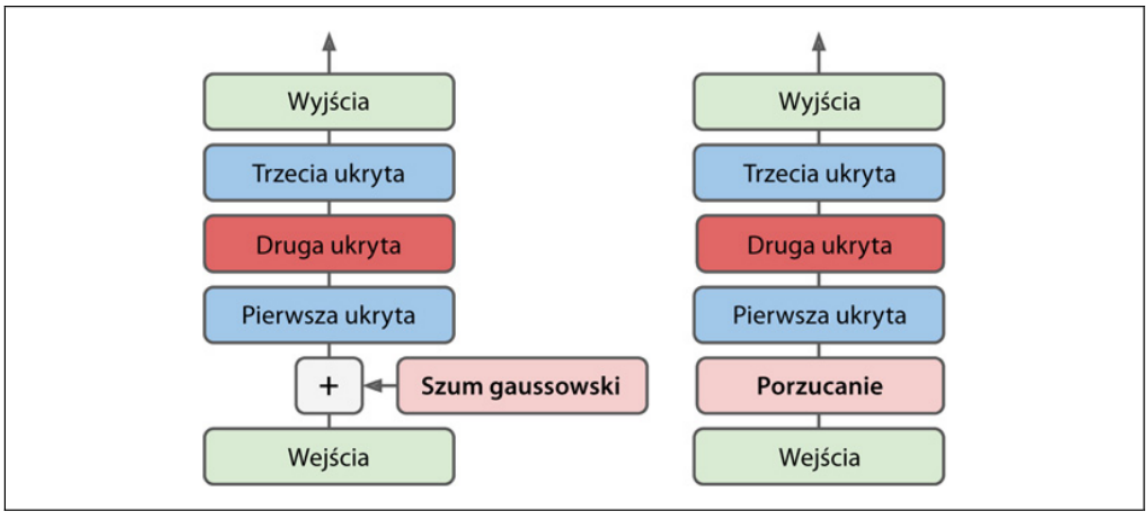
1.4.6. Autoenkodey odszumiające

[6]

Kolejną metodą zmuszania autokodera do poznawania przydatnych cech jest dodawanie szumu do danych wejściowych i uczenie go odzyskiwania pierwotnych, niezaszumionych informacji. Pierw- sze koncepcje wykorzystania autokoderów do odszumiania danych pojawiły się już w latach 80. ubie- głego wieku (m.in. pomysł ten został wspomniany w pracy magisterskiej Yanna LeCuna w 1987 roku). Pascal Vincent i in. udowodnili w publikacji z 2008 roku (https://www.iro.umontreal.ca/ vincentp/ Publications/denoising_autoencoders_tr1316.pdf) 5 , że autokodery mogą być również używane do wydobywania cech. Z kolei ten sam autor i in. w artykule z 2010 roku (http://jmlr.csail.mit.edu/ papers/v11/vincent10a) 6 zaprezentowali odszumiające autokodery stosowe (ang. stacked denoising autoencoders). Zaszumienie może być standardowym szumem gaussowskim dodawanym do danych wejściowych lub może przybrać postać losowo wyłączanych wejść za pomocą metody porzucania (zob. rozdział 11.). Obydwa rozwiązania zostały pokazane na rysunku 17.8. Implementacja nie stanowi wyzwania: jest to standardowy autokoder stosowy zawierający dodatkową warstwę Dropout , przez którą przechodzą dane wejściowe (możesz zastąpić ją warstwą GaussianNoise). Jak pamiętamy, warstwa Dropout jest aktywna jedynie w fazie uczenia (podobnie jak warstwa GaussianNoise):

```
dropout_encoder = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.Dropout(0.5),
keras.layers.Dense(100, activation="selu"),
keras.layers.Dense(30, activation="selu")
])
```

```
dropout_decoder = keras.models.Sequential([
keras.layers.Dense(100, activation="selu", input_shape=[30]),
keras.layers.Dense(28 * 28, activation="sigmoid"),
keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```



Rysunek 1.16: Autokodery odszumiające: wykorzystujące szum gaussowski (po lewej) lub metodę porzucania (po prawej)

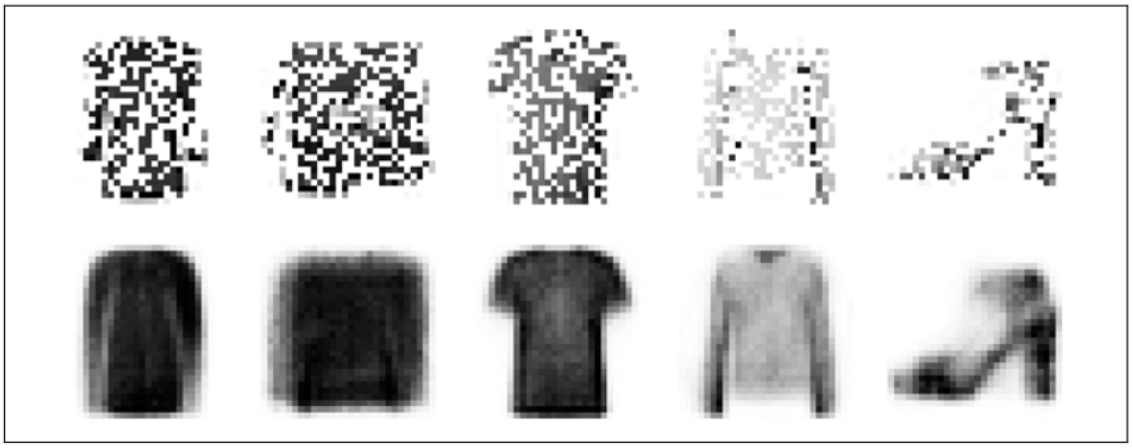
Źródło: [6]

Rysunek 17.9 przedstawia kilka zaszumionych obrazów (połowa pikseli została „wyłączona”), a także ich rekonstrukcje uzyskane za pomocą autokodera odszumiającego (bazującego na warstwie po- rzucania). Zwróć uwagę, że autokoder w istocie „zgaduje” szczegóły niewystępujące w obrazach wej- ściowych, na przykład górną część białej sukienki (czwarty obraz w dolnym rzędzie). Jak widać, autokodery odszumiające służą nie tylko do wizualizowania danych lub nienadzorowanego uczenia wstępnego, lecz również w dość prosty i skuteczny sposób mogą usuwać szum z obrazów.

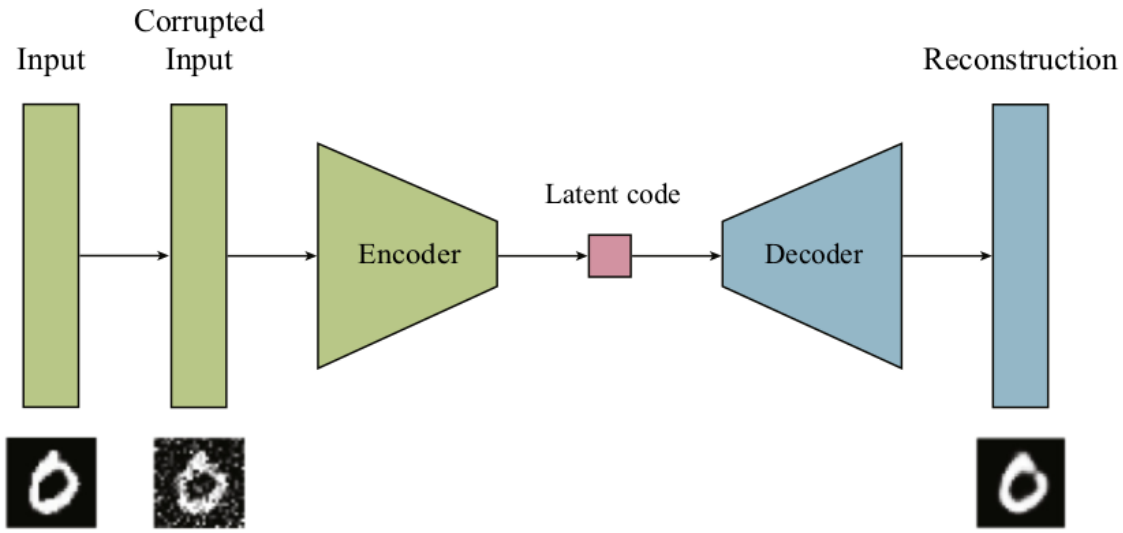
1.4.7. Autoenkodery rzadkie

[6]

Innym ograniczeniem prowadzącym do skutecznego wydobywania cech jest rzadkość (ang. sparsity): przez dodanie odpowiedniego członu do funkcji kosztu autokoder zostaje zmuszony do zmniejszenia liczby aktywnych neuronów w warstwie kodowania. Możemy sprawić na przykład, żeby w warstwie kodującej występowało tylko 5% znacząco aktywnych neuronów. W ten sposób wymuszamy na auto- koderze reprezentowanie każdego wejścia



Rysunek 1.17: Zaszumione obrazy (na górze) i ich rekonstrukcje (na dole)
Źródło: [6]



Rysunek 1.18: The structure of the denoising autoencoder. Here, the input to the autoen- coder is the noisy data, whereas the expected target is the original noise-free data. The network learns to recognize and remove the noise in the input data before generating the output. The autoencoder does not see the original, noise-free image at all, it only sees the image that has undergone some corruption. This corruption process is flexible; it can be based on masking noise, Gaussian noise, salt-and-pepper noise (shown in the example), or even prior knowledge.
Źródło: [12]

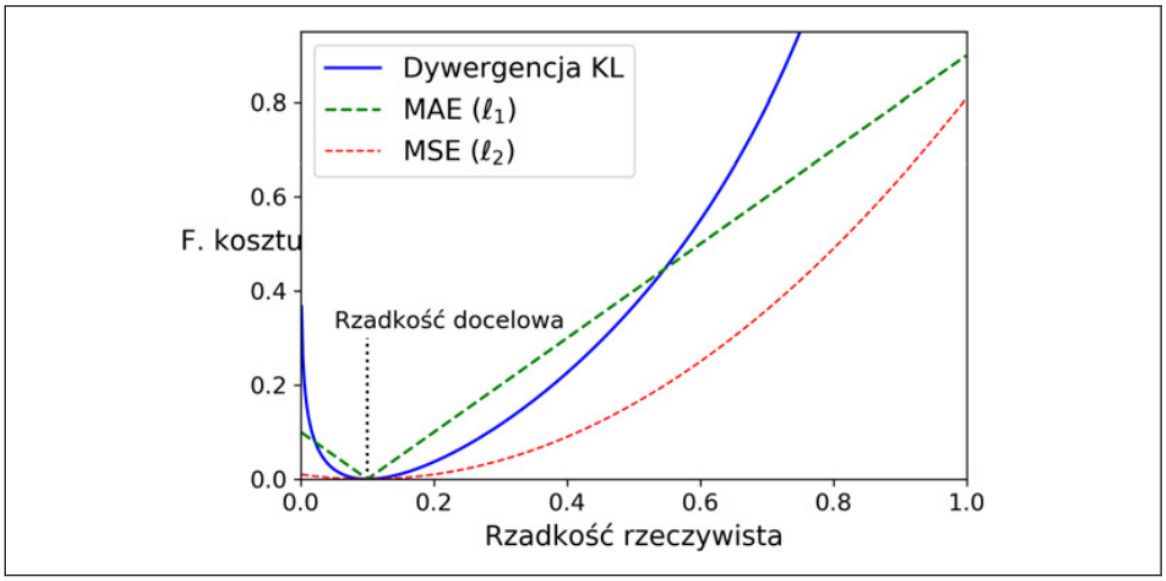
jako kombinacji niewielkiej liczby pobudzeń. Dzięki temu każdy neuron warstwy kodowania zazwyczaj uczy się wykrywać jakąś przydatną cechę (gdybyśmy wypowiadali w ciągu miesiąca tylko kilka słów, prawdopodobnie chcielibyśmy przekazywać za ich pomocą wartościowe informacje). Prostym rozwiązaniem okazuje się wprowadzenie sigmoidalnej funkcji aktywacji w warstwie kodowa- nia (co ogranicza wartości kodowań do zakresu od 0 do 1), wprowadzenie dużej warstwy kodowania (np. składającej się z 300 jednostek) i dodanie re-

gularyzacji l 1 do pobudzeń warstwy kodowania (w dekoderze nie wprowadzamy żadnych zmian):

```
sparse_l1_encoder = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.Dense(100, activation="selu"),
keras.layers.Dense(300, activation="sigmoid"),
keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_l1_decoder = keras.models.Sequential([
keras.layers.Dense(100, activation="selu", input_shape=[300]),
keras.layers.Dense(28 * 28, activation="sigmoid"),
keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder,
sparse_l1_decoder])
```

Taka warstwa ActivityRegularization zwraca jedynie swoje dane wejściowe, ale jako skutek uboczny dodaje funkcję straty uczenia równą sumie wartości bezwzględnych tychże sygnałów wejściowych (warstwa ta jest aktywna wyłącznie w fazie uczenia). Ewentualnie możesz usunąć warstwę Activity Regularization i wyznaczyć atrybut activity_regularizer= keras.regularizers.l1(1e-3) w war- stwie poprzedzającej. Kara ta będzie zmuszała sieć neuronową do tworzenia kodowań o wartościach bliskich 0, ale jednocześnie model będzie również karany za niewłaściwe rekonstruowanie danych wejściowych, więc będzie musiał wyznaczać co najmniej kilka wartości niezerowych. Norma l 1 sprawia, że sieć będzie przechowywała najważniejsze kodowania i eliminowała nieprzydatne z per- spektywy obrazu wejściowego (w przeciwieństwie do normy l 2 , która jedynie redukowałaby wszystkie kodowania). Innym rozwiązaniem, często dającym lepsze rezultaty, jest pomiar rzeczywistej rzadkości warstwy ko- dowania w każdym przebiegu uczenia i karanie modelu w sytuacji, gdy zmierzona rzadkość różni się od rzadkości docelowej. Robimy to, obliczając średnią aktywację każdego neuronu w tej war- stwie dla całej grupy próbek uczących. Rozmiar tej grupy nie może być zbyt mały, gdyż wyliczona wartość średniej nie będzie wtedy dokładna. Po uzyskaniu średniej wartości aktywacji każdego neuronu chcemy nałożyć karę na zbyt aktywne neurony poprzez dodanie funkcji straty rzadkości (ang. sparsity loss) do funkcji kosztu. Przykładowo jeśli zmierzymy średnią wartość aktywacji neuronu rzędu 0,3, ale aktywność docelowa powinna wyno- sić 0,1, musimy zmniejszyć jego aktywność. Jednym ze sposobów jest dodanie kwadratu błędu (0,3-0,1) 2 do funkcji kosztu, w praktyce jednak lepszym rozwiązaniem okazuje

się wprowadzenie dy- wergencji Kullbacka-Leiblera (zob. rozdział 4.), której gradienty są znacznie większe niż w błędzie średniokwadratowym (rysunek 17.10).



Rysunek 1.19: Funkcje straty rzadkości

Źródło: [6]

Mając dwa dyskretne rozkłady prawdopodobieństwa P i Q, możemy obliczyć rozbieżność pomiędzy nimi $D_{KL}(P||Q)$ za pomocą równania 17.1. (dywergencja Kullbacka-Leiblera)

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

W naszym przypadku chcemy zmierzyć rozbieżność pomiędzy docelowym prawdopodobieństwem p aktywacji neuronu w warstwie kodowania a rzeczywistym prawdopodobieństwem q (tzn. średnią aktywacją dla grupy przykładów uczących). Zatem dywergencja KL ulega uproszczeniu do postaci widocznej w równaniu 17.2. (Dywergencja KL pomiędzy docelową rzadkością p a rzeczywistą rzadkością q)

$$D_{KL}(p||q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

Po obliczeniu funkcji straty rzadkości dla każdego neuronu w warstwie kodowania wystarczy je zsu- mować i dodać wynik do funkcji kosztu. W celu regulowania względnej istotności funkcji straty rzadkości i funkcji straty rekonstrukcji możemy pomnożyć tę pierwszą przez hiperparametr wagi rzadkości. Jeżeli wartość wagi będzie za duża, model pozostanie blisko rzadkości docelowej, ale jednocześnie może nie być w stanie prawidłowo rekonstruować danych wejściowych, przez co okaże się bezużyteczny. Z kolei przy zbyt małej wartości wagi rzadkości model będzie ignorował cel rzadkości i nie nauczy się rozpoznawać przydatnych cech.

1.5. Zastosowania autoenkoderów

https://towardsdatascience.com/6-applications-of-auto-encoders-every-data-scient

Autoenkodery są popularnym rodzajem nienadzorowanej sztucznej sieci neuronowej, która na wejściu korzysta z nieoznakowanych danych i uczy się efektywnego kodowania struktury tych danych, które może być użyte w kontekście innych zadań. Autoenkodery aproksymują funkcję, która odwzorowuje dane z pełnej przestrzeni wejściowej na współrzędne niższego wymiaru, i dalej aproksymuje do tego samego wymiaru przestrzeni wejściowej z minimalną stratą.

W zadaniach klasyfikacji lub regresji auto-enkodery mogą być wykorzystywane do wyodrębniania cech z surowych danych w celu zwiększenia odporności modelu. Istnieje wiele innych zastosowań sieci auto-enkoderów, które można wykorzystać w innym kontekście. W tym artykule omówimy 7 takich zastosowań auto-enkodera:

1) Dimensionality Reduction (zmniejszenie wymiarowości)

2) Feature Extraction (wyodrębnianie cech)

3) Image Denoising (odszumianie obrazu)

4) Image Compression (kompresja obrazu)

5) Image Search (wyszukiwanie obrazu)

6) Anomaly Detection (wykrywanie anomalii)

7) Missing Value Imputation (uzupełnianie brakujących danych)

1.5.1. Zmniejszenie wymiarowości

(źródło jak wyżej, ale również tu: https://towardsdatascience.com/autoencoders-in-practice

Autoenkodery trenują sieć w celu wyjaśnienia naturalnej struktury danych w efektywnej reprezentacji niskowymiarowej. Osiąga się to poprzez zastosowanie strategii dekodowania i kodowania w celu zminimalizowania błędu rekonstrukcji.

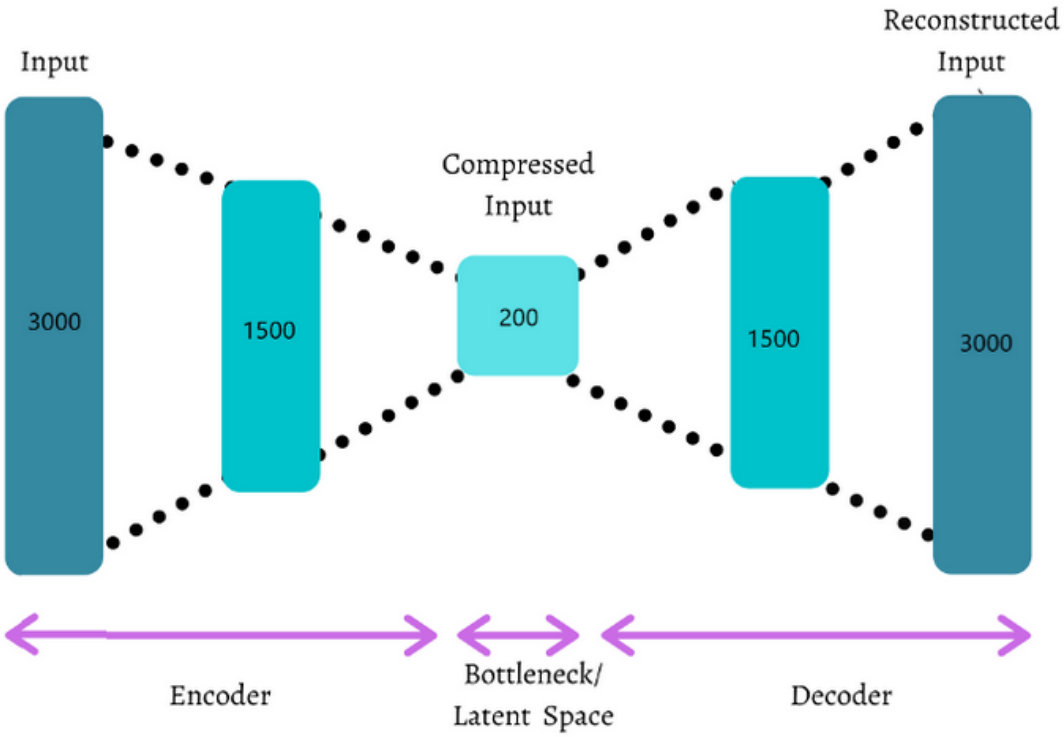
Jak widać powyżej, autoenkoder składa się z trzech elementów:

Koder - funkcja służąca do kompresji danych do ich reprezentacji w niższym wymiarze.

Wąskie gardło - nazywane również przestrzenią ukrytą, gdzie nasze początkowe dane są reprezentowane w niższym wymiarze.

Dekoder - funkcja dekompresji lub rekonstrukcji danych o niskim wymiarze z powrotem do wymiaru początkowego.

Wymiar wejściowy i wymiar wyjściowy mają 3000 wymiarów, a pożądany wymiar zredukowany wynosi 200. Możemy stworzyć sieć pięciowarstwową, w której koder ma 3000 i 1500 neuronów, podobnie jak w przypadku sieci dekodera.



Rysunek 1.20: Redukcja wymiarowości
Źródło: towards data science

Embeddingi wektorowe skompresowanej warstwy wejściowej można traktować jako embedding warstwy wejściowej o zredukowanym wymiarze.

1.5.2. Wyodrębnianie cech

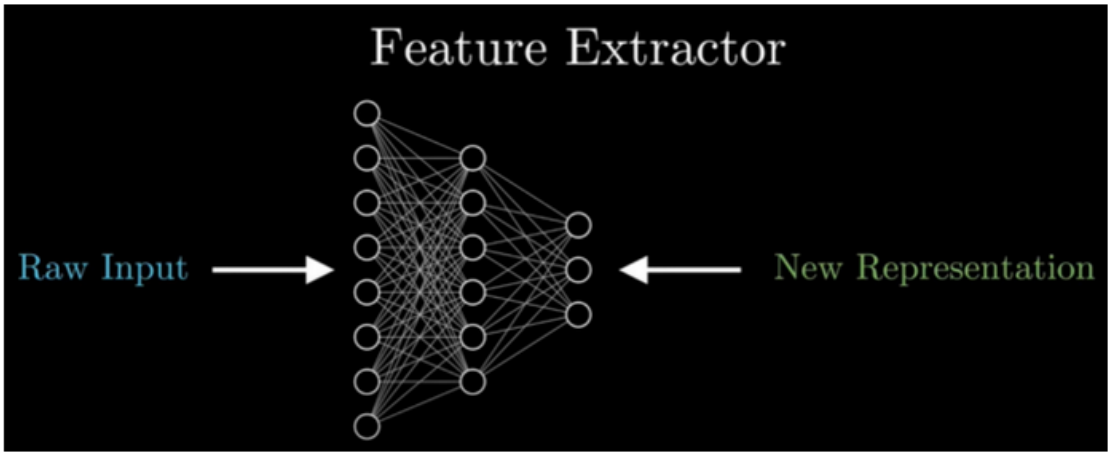
Autokodery mogą być używane jako ekstraktory cech w zadaniach klasyfikacji lub regresji. Autokodery pobierają nieoznakowane dane i uczą się efektywnych kodowań struktury danych, które mogą być wykorzystane w zadaniach uczenia nadzorowanego.

Po wytrenowaniu sieci auto-enkodera na próbce danych treningowych można zignorować część dekodera auto-enkodera, a jedynie użyć kodera do przekształcenia surowych danych wejściowych o wyższym wymiarze na przestrzeń zakodowaną o niższym wymiarze. Ten niższy wymiar danych może być wykorzystany jako cecha w zadaniach nadzorowanych.

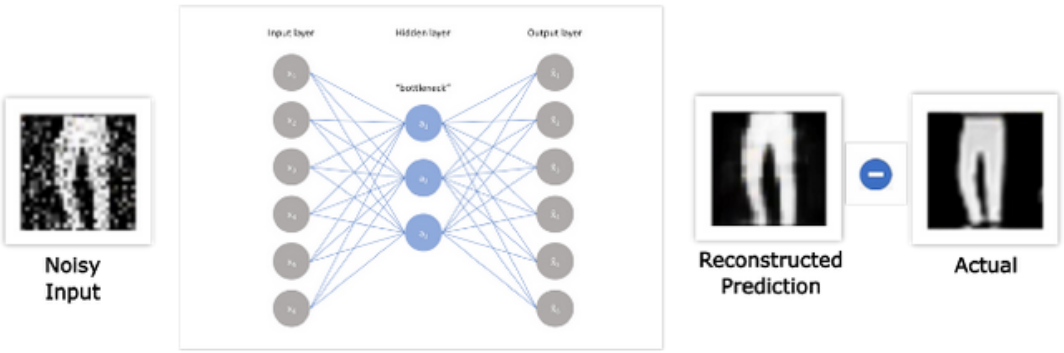
1.5.3. Odszumianie obrazów

Surowe dane wejściowe ze świata rzeczywistego są często zaszumione, a wytrenowanie solidnego modelu nadzorowanego wymaga danych oczyszczonych i pozbawionych szumu. Do odszumiania danych można wykorzystać autoenkodery.

Jednym z popularnych zastosowań jest odszumianie obrazów, w którym autoenkodery próbują zrekonstruować obraz pozbawiony szumu z zaszumionego obrazu wejściowego.



Rysunek 1.21: Wyodrębnianie cech
Źródło: towards data science



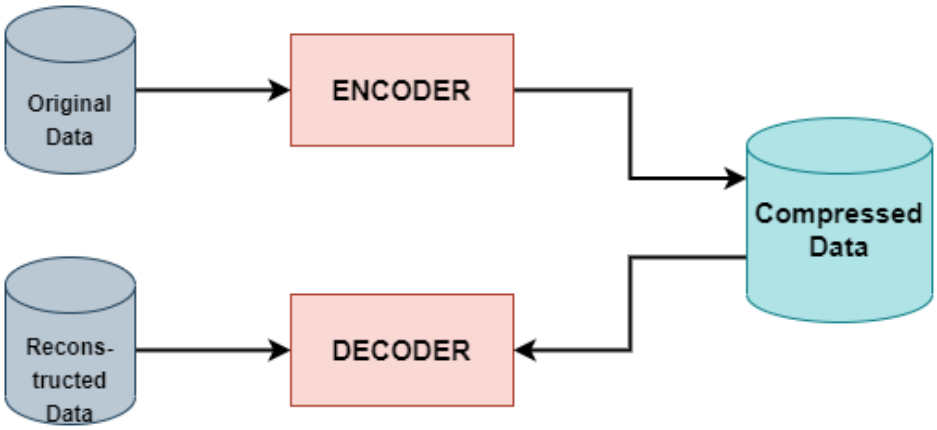
Rysunek 1.22: Odszumianie obrazów
Źródło: towards data science

Zakłócony obraz wejściowy jest podawany do autoenkodera jako wejście, a bezszumowe wyjście jest rekonstruowane przez minimalizację straty rekonstrukcji w stosunku do oryginalnego wyjścia docelowego (bezszumowego). Po wytrenowaniu wag autoenkodera można je dalej wykorzystać do odszumiania obrazu surowego.

1.5.4. Kompresja obrazu

Innym zastosowaniem sieci auto-enkoderów jest kompresja obrazów. Surowy obraz wejściowy można przekazać do sieci kodera i uzyskać skompresowany wymiar zakodowanych danych. Wagi sieci autoenkodera mogą być uczone przez rekonstrukcję obrazu ze skompresowanego kodowania za pomocą sieci dekodera.

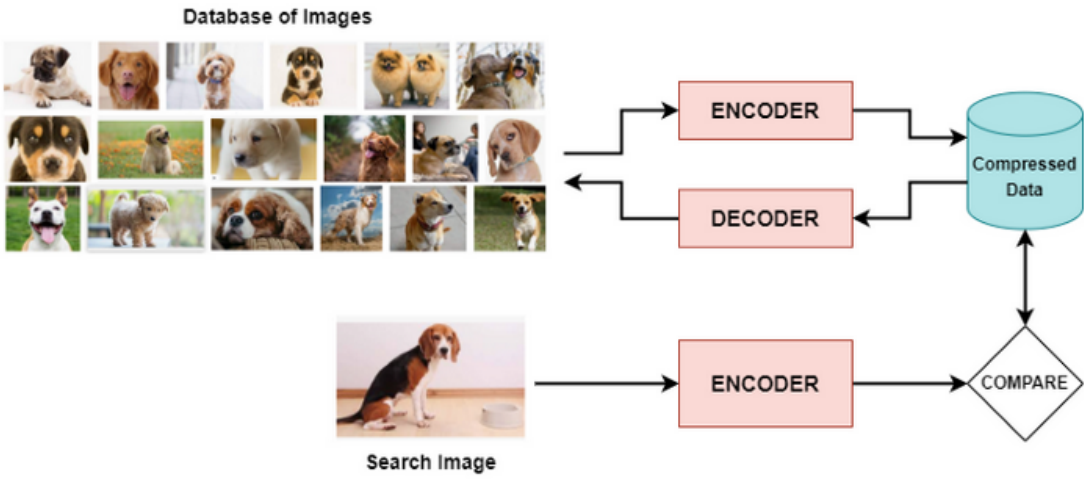
Zazwyczaj autoenkodery nie nadają się zbyt dobrze do kompresji danych, lepiej sprawdzają się raczej podstawowe algorytmy kompresji.



Rysunek 1.23: kompresja obrazów
Źródło: towards data science

1.5.5. Szukanie obrazu

Do kompresji bazy danych obrazów można użyć autokoderów. Skompresowane osadzenie może być porównywane lub przeszukiwane z zakodowaną wersją szukanego obrazu.



Rysunek 1.24: szukanie obrazów
Źródło: towards data science

1.5.6. Wykrywanie anomalii

Innym użytecznym zastosowaniem sieci autoenkoderów jest wykrywanie anomalii. Model wykrywania anomalii można wykorzystać do wykrywania oszukańczych transakcji lub wszelkich zadań nadzorowanych o wysokim stopniu nierównowagi.

Idea polega na trenowaniu autoenkoderów tylko na próbkach danych jednej klasy (klasy większościowej). W ten sposób sieć jest w stanie zrekonstruować dane wejściowe z dobrą

lub mniejszą stratą rekonstrukcji. Jeśli przez sieć autoenkodera przepuści się próbkę danych innej klasy docelowej, spowoduje to porównywalnie większą stratę rekonstrukcji.

Można określić wartość progową straty rekonstrukcji (wynik anomalii), której przekroczenie będzie uznawane za anomalię.

1.5.7. Uzupełnianie brakujących danych

Do imputacji brakujących wartości w zbiorze danych można wykorzystać autoenkodery odszumiające. Idea polega na trenowaniu sieci autoenkoderów poprzez losowe umieszczanie brakujących wartości w danych wejściowych i próbę odtworzenia oryginalnych danych surowych poprzez minimalizację straty rekonstrukcji.

Po wytrenowaniu wag autoenkodera rekordy zawierające brakujące wartości mogą być przepuszczane przez sieć autoenkodera w celu zrekonstruowania danych wejściowych, także z imputowanymi brakującymi cechami.

Rozdział 2

Przykłady zastosowań autoenkoderów

2.1. Analiza PCA za pomocą autoenkodera niedopełnionego

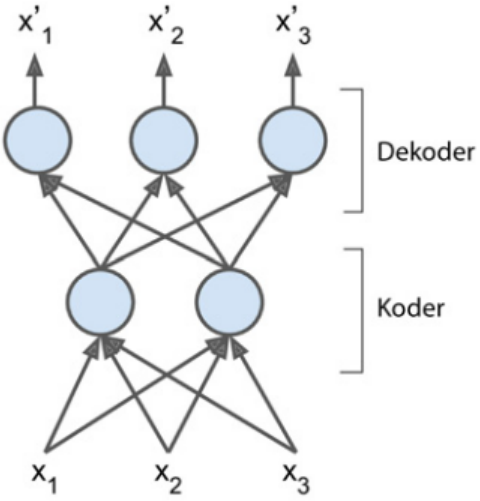
Jeśli autoenkoder korzysta jedynie z liniowych funkcji aktywacji, a funkcją kosztu jest błąd średniokwadratowy (MSE), to będzie on przeprowadzał analizę składowych głównych PCA [6]. Na rysunku 2.1 widoczny jest kod w języku Python służący do utworzenia takiego autoenkodera, który przeprowadzi analizę PCA na zbiorze trójwymiarowym, rzutując go na przestrzeń dwuwymiarową. Można zauważyć podział autoenkodera na dwie części - enkoder i dekoder. Enkoder przyjmuje dane wejściowe o wymiarze 3, a na wyjściu pojawiają się dane dwuwymiarowe. W przypadku dekodera jest odwrotnie - dane na wejściu są dwuwymiarowe, a na wyjściu trójwymiarowe. Obydwie części są modelami sekwencyjnymi z jedną warstwą gęstą, a cały autoenkoder jest modelem sekwencyjnym, w którym po enkoderze występuje dekoder.

```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(learning_rate=1.5))
```

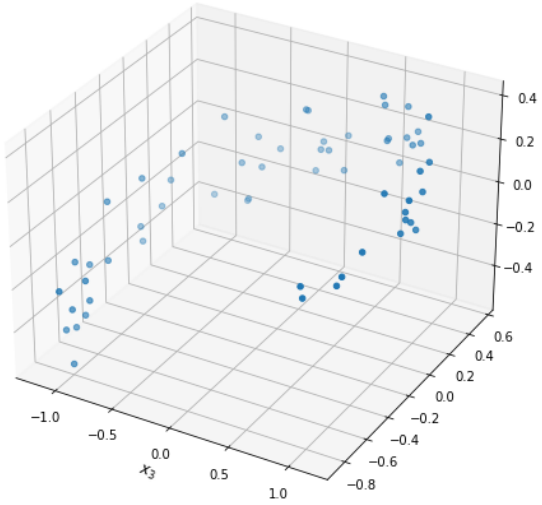
Rysunek 2.1: Tworzenie autoenkodera niedopełnionego przeprowadzającego PCA

Źródło: Opracowanie własne na podstawie [6]



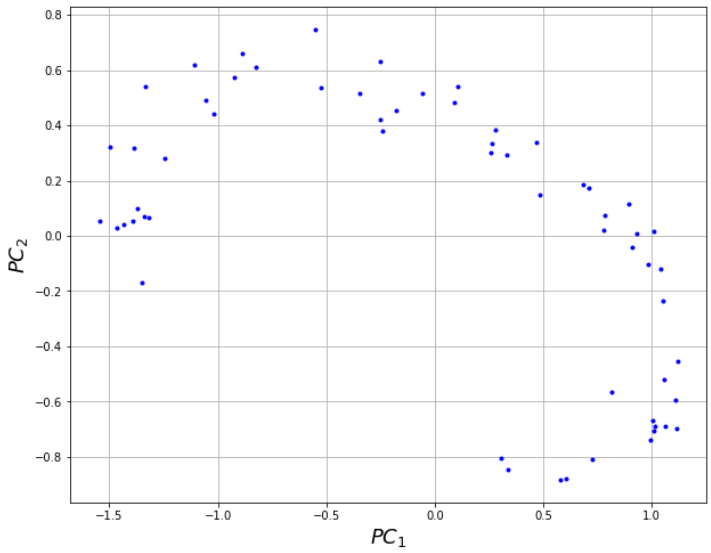
Rysunek 2.2: Architektura autoenkodera niedopełnionego
Źródło: [6]

Opisany autoenkoder zostanie wyuczony na wygenerowanym zbiorze trójwymiarowych danych. Warto zwrócić uwagę, że zbiór uczący stanowi jednocześnie dane wejściowe i dane docelowe. Po wytrenowaniu modelu, używamy enkodera do zakodowania danych wejściowych, czyli do zrzutowania ich na przestrzeń dwuwymiarową. Na rysunku 2.3 zaprezentowany jest trójwymiarowy zbiór danych. Rysunek 2.4 przedstawia wynik działania autoenkodera, a więc dane zrzutowane na przestrzeń 2D. Z kolei na rysunku 2.5 widoczne jest rzutowanie uzyskane przy użyciu „zwykłego" algorytmu PCA. Na podstawie tych wykresów można stwierdzić, że rzutowanie uzyskane przy pomocy autoenkodera oraz PCA jest takie samo, jedynie układ współrzędnych jest odwrócony o 180 stopni.



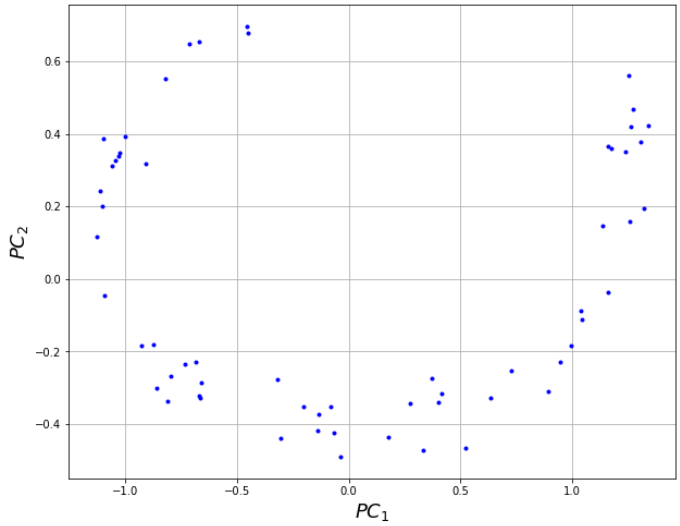
Rysunek 2.3: Wygenerowane dane trójwymiarowe

Źródło: Opracowanie własne na podstawie [6]

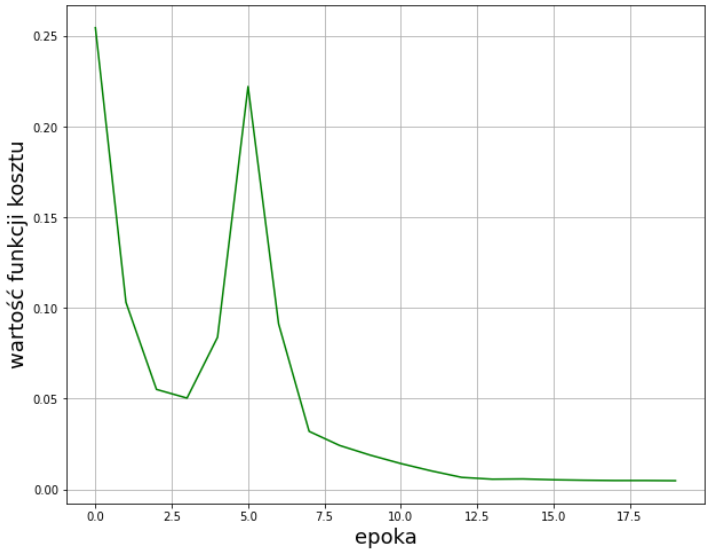


Rysunek 2.4: Rzutowanie dwuwymiarowe przy użyciu autoenkodera, zachowujące maksymalną wariancję

Źródło: Opracowanie własne na podstawie [6]



Rysunek 2.5: Rzutowanie dwuwymiarowe przy użyciu PCA, zachowujące maksymalną wariancję
Źródło: Opracowanie własne na podstawie [6]



Rysunek 2.6: Wartość funkcji straty w kolejnych epokach podczas uczenia modelu
Źródło: Opracowanie własne

2.2. Kolejny przykład

Podsumowanie i wnioski

Bibliografia

[1] Aggarwal, C. C. (2018). *Neural Networks and Deep Learning*, Springer International Publishing

[2] Bank D., Koenigstein N., Giryes R. (2021), *Autoencoders*, arXiv:2003.05991v2

[3] Chollet F., Allaire J. J. (2019) *Deep Learning. Praca z językiem R i biblioteką Keras*, Helion SA

[4] Edureka, Autoencoders Tutorial. Autoencoders in Deep Learning. Tensorflow Training `https://www.youtube.com/watch?v=nTt_ajul8NY`

[5] Ertel W. (2017) *Introduction to Artificial Intelligence. Second Edition*, Springer International Publishing

[6] Géron A. (2020) *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow. Wydanie II*, Helion SA

[7] Goodfellow I., Bengio Y., Courville A. (2018), *Deep Learning. Systemy uczące się*, PWN, Warszawa

[8] Hubel D. H., *Single Unit Activity in Striate Cortex of Unrestrained Cats*, J. Physiol. (1959) 147, 226-238

[9] Hubel D. H., Wiesel T. N., *Receptive Fields of Single Neurones in the Cat's Striate Cortex*, J. Physiol., (1959), 148, 574-591

[10] Krohn J., Beyleveld G., Bassens A., *Uczenie głębokie i sztuczna inteligencja. Interaktywny przewodnik ilustrowany*, Helion 2022

[11] Osinga D. (2019) *Deep Learning. Receptury*, Helion SA

[12] Pinaya W. H. L., Vieira S., Garcia-Dias R., Mechelli A., *Machine Learning. Methods and Applications to Brain Disorders*, Academic Press, 2020

[13] Skansi S. (2018) *Introduction to Deep Learning. From Logical Calculus to Artificial Intelligence*, Springer International Publishing

Spis rysunków

Spis tabel

Załączniki

1. Płyta CD z niniejszą pracą w wersji elektronicznej.

Streszczenie (Summary)

Przegląd autoenkoderów stosowanych w nienadzorowanym uczeniu maszynowym

An overview of autoencoders used in unsupervised machine learning