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

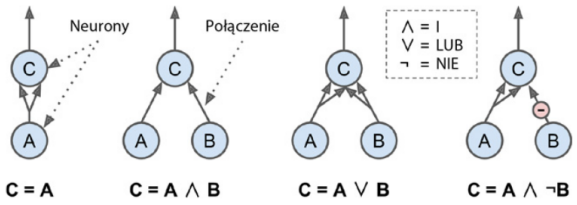
Wstęp

Rozdział 1

Przegląd autoenkoderów

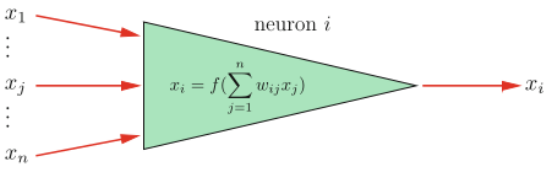
1.1. Sztuczne sieci neuronowe

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]

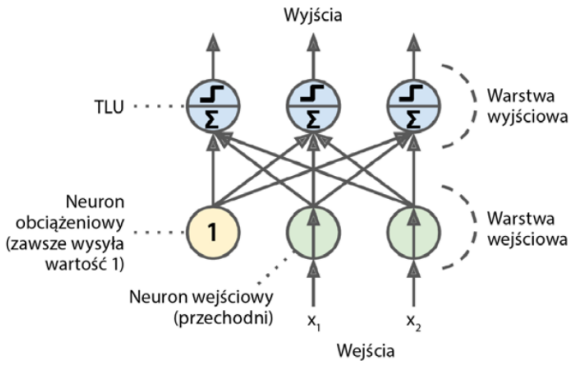
Często używaną funkcją skokową jest **funkcja Heaviside'a**, określona równaniem

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

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 [10]. Można go zapisać jako

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

gdzie E jest funkcją kosztu (funkcją straty) [10]. 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.

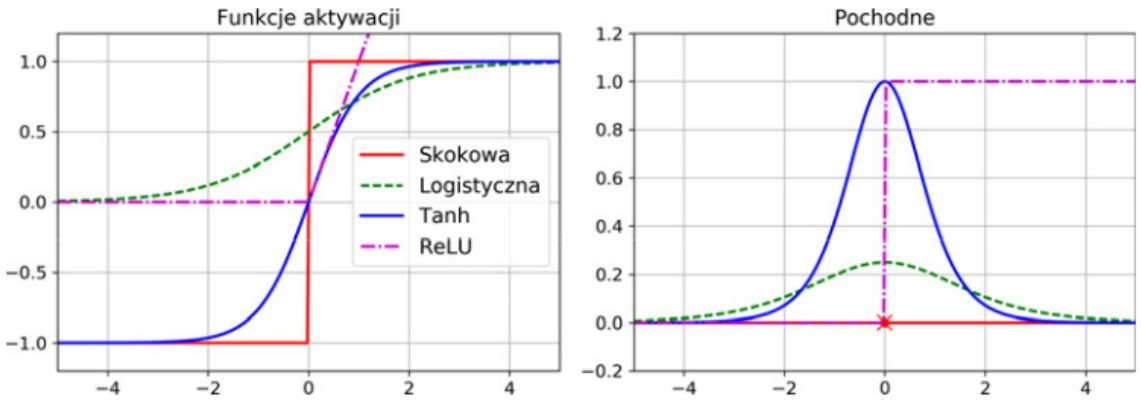
Wśród popularnych funkcji aktywacji należy także wyróżnić **funkcję ReLU** (ang. *Rectified Linear Unit* - prostowana jednostka liniowa) o wzorze

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

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

Sieci splotowe, nazywane również **splotowymi sieciami neuronowymi** (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].

[6]

Splotowe sieci neuronowe stanowią wynik badań nad korą wzrokową. Są używane głównie do rozpoznawania obrazów (od lat 80-tych XX w.).

Zagadnienia rozpoznawania obrazu:

— klasyfikacja
— detekcja obrazów
— transfer stylu

Neurony biologiczne w korze wzrokowej reagują na określone wzorce w niewielkich obszarach pola wzrokowego, zwanych polami recepcyjnymi; w miarę przepływu sygnału wzrokowego przez kolejne moduły w mózgu neurony rozpoznają coraz bardziej skomplikowane wzorce wykrywane w coraz większych polach recepcyjnych.

Wiele neuronów tworzących korę wzrokową tworzą lokalne pola recepcyjne, reagujące jedynie na bodźce wzrokowe mieszczące się w określonym rejonie pola wzrokowego.

Pola recepcyjne poszczególnych neuronów mogą się na siebie nakładać.

Pola recepcyjne łącznie tworzą całe pole wzrokowe.

Pewne neurony reagują wyłącznie na obrazy składające się z linii poziomych, lub innych linii ułożonych w konkretny sposób.

Niektóre komórki nerwowe mają większe pola recepcyjne i wykrywają bardziej skomplikowane kształty.

Neurony odpowiedzialne za rozpoznawanie bardziej skomplikowanych kształtów znajdują się na wyjściu neuronów reagujących na prostsze bodźce.

1.2.1. Hiperparametry filtru konwolucyjnego

[8]

Warstwy konwolucyjne w przeciwieństwie do zagęszczonych, nie są w pełni połączone. Oznacza to, że w pierwszej warstwie ukrytej nie są wiązane wagi wszystkich pikseli ze

wszystkimi neuronami. Zamiast tego stosowanych jest kilka wymienionych niżej hiperparametrów określających liczbę wag i odchyleń związanych z daną warstwą konwolucyjną:

— wielkość jądra (kernel size) - 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. Popularna 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.

— 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.

— dopełnienie (padding) - jest ściśle związane z długością kroku. Zapewnia poprawność obliczeń w warstwie konwolucyjnej.

1.2.2. Warstwy redukujące

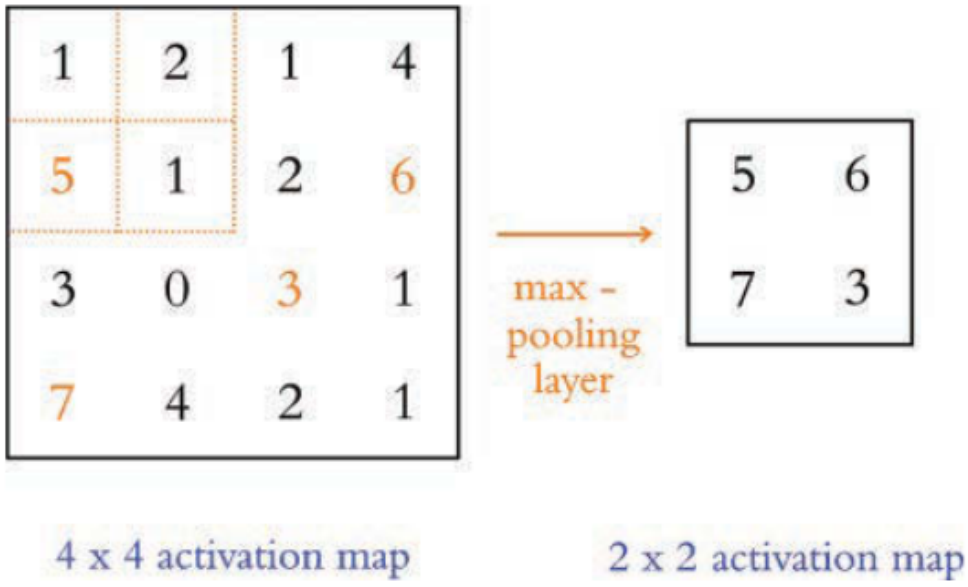
[8]

Warstwy konwolucyjne często współpracują z innego typu warstwami, stanowiącymi fundament sieci neuronowych i widzenia maszynowego. Są to warstwy redukujące (pooling layers), których zadaniem jest zmniejszanie ogólnej liczby parametrów sieci, redukowanie złożoności, przyspieszanie obliczeń i zapobieganie nadmiernemu dopasowaniu modelu.

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.

Z warstwą redukującą, podobnie jak z konwolucyjną, jest związany filtr i długość kroku. Filtr przesuwa się nad danymi wejściowymi, tak jak w warstwie konwolucyjnej, i w każdej zajmowanej pozycji przeprowadza operację redukcji danych. Najczęściej jest to wybieranie największej wartości, dlatego taka warstwa jest również określana mianem maksymalnie redukującej (max-pooling layer). Z całego pola receptywnego jest wybierana największa wartość (maksymalna aktywacja), a pozostałe wartości są odrzucane. 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. Ponieważ operacja redukcji jest wykonywana niezależnie w każdym wycinku trójwymiarowej tablicy, mapa aktywacji o wymiarach 28x28 elementów i głębokości 16 wycinków jest redukowana do tablicy 14x14 elementów. Zachowywana jest jednak jej pełna głębokość 16 wycinków.



Rysunek 1.5: Max polling
Źródło: [8]

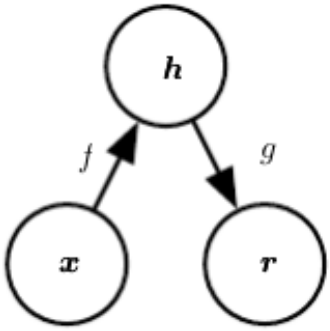
1.3. Czym jest autoenkoder

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, rozbudowane-

go 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.



Rysunek 1.6: 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]

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

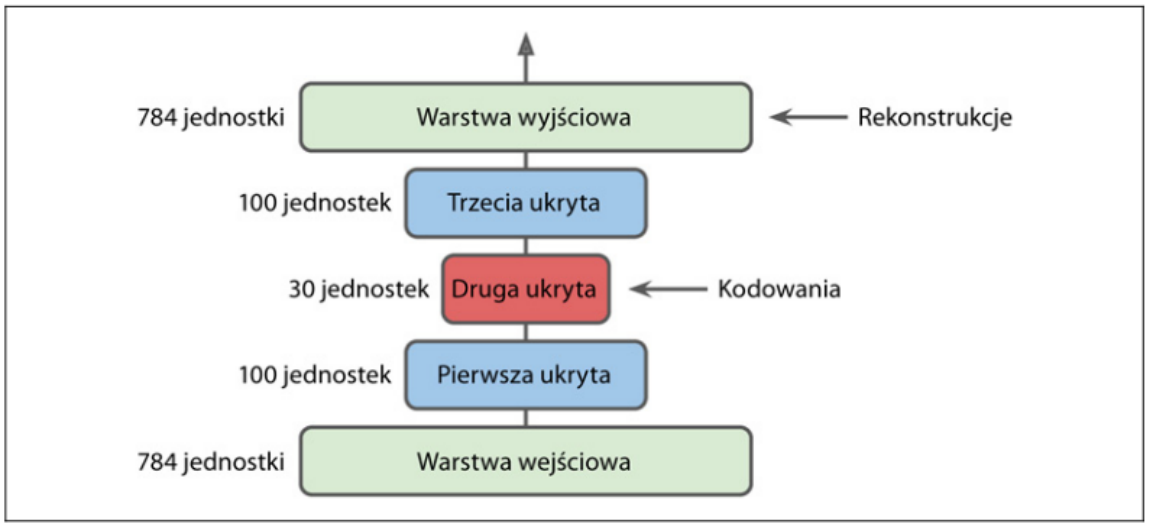
$$L(x, g(f(x)))$$

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.

1.4.2. 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.



Rysunek 1.7: Stosowy
Źródło: [6]

1.4.3. 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).

1.4.4. 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.

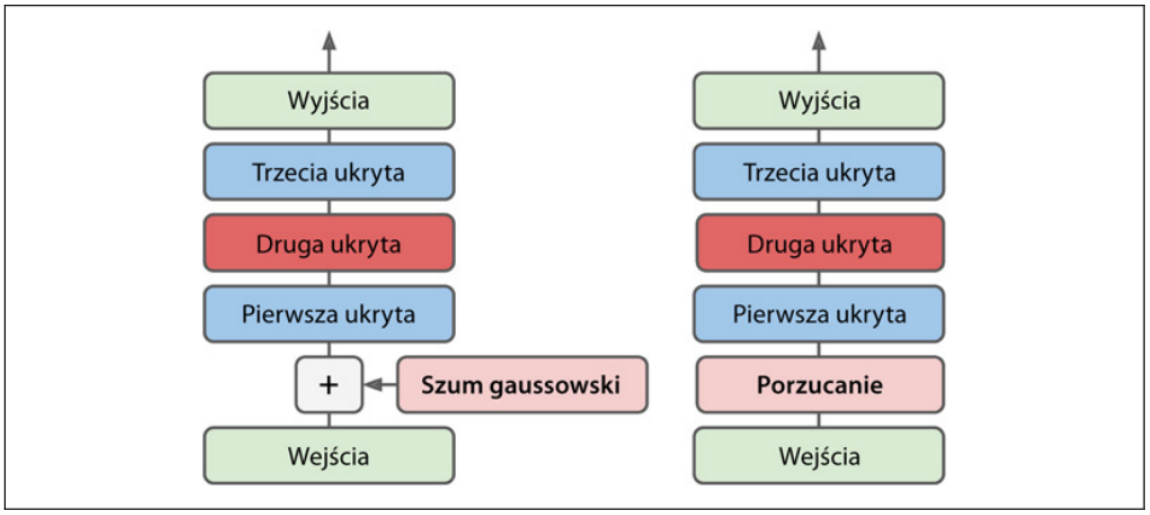
1.4.5. Autoenkodey odszumiające

[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):

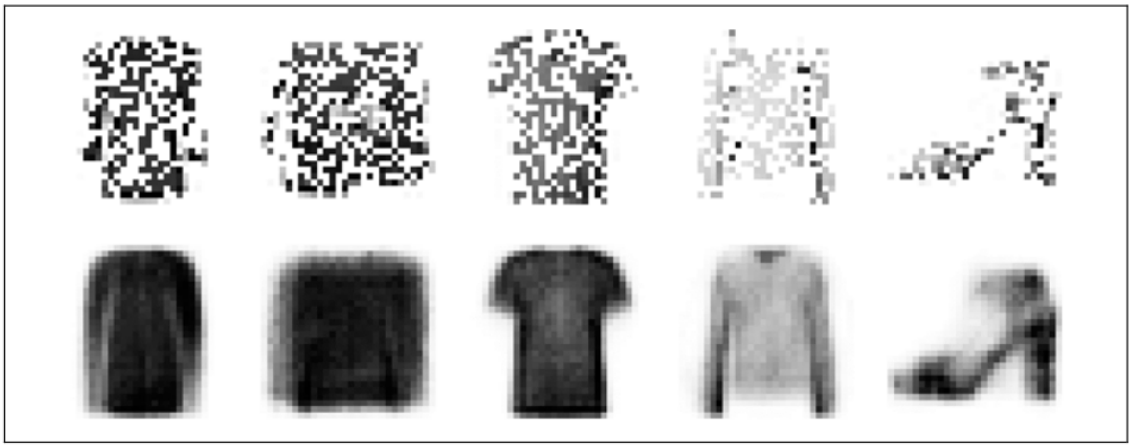
```
dropout_encoder = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.Dropout(0.5),
keras.layers.Dense(100, activation="selu"),
keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
keras.layers.Dense(100, activation="selu", input_shape=[30]),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



Rysunek 1.8: 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.



Rysunek 1.9: Zaszumione obrazy (na górze) i ich rekonstrukcje (na dole)
Źródło: [6]

1.4.6. 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 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 regularyzacji 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"),
```

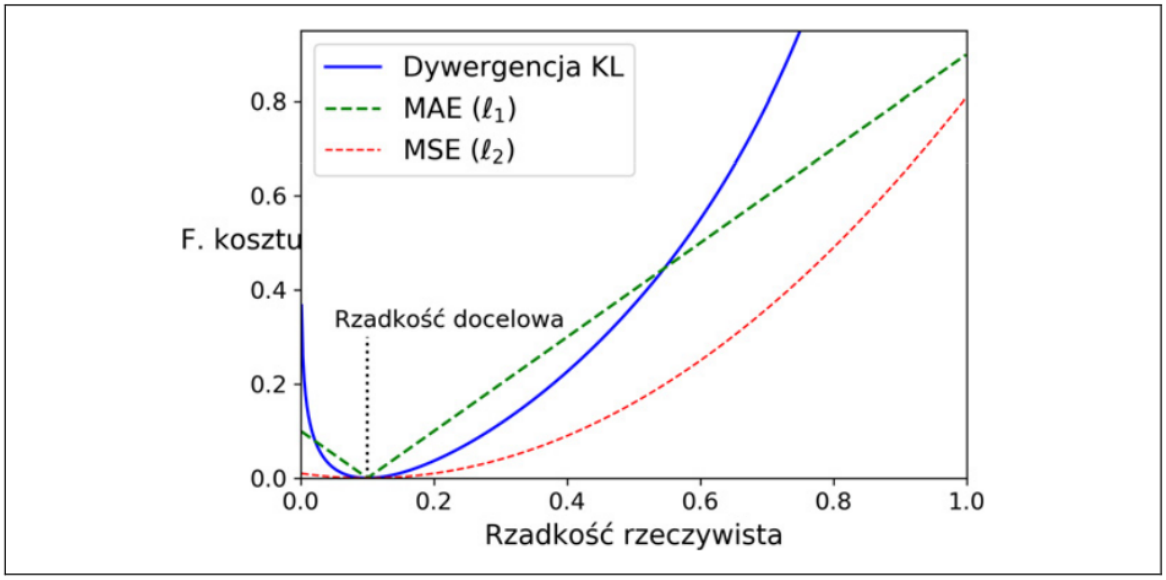
```
keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder,
sparse_l1_decoder])
```

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).

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)



Rysunek 1.10: Funkcje straty rzadkości

Źródło: [6]

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

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.

Rozdział 2

Przykłady zastosowań autoenkoderów

2.1. Analiza PCA za pomocą autoenkodera niedopełnionego

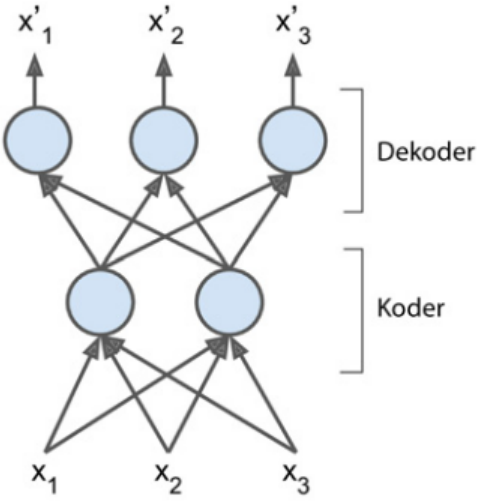
Jeśli autoenkoder korzysta jedynie z liniowych funkcji aktywacji, a funkcją kosztu jest błąd średniokwadratowy (MSE), to będzie on przeprowadzał analizę składowych głównych PCA [6]. Na rysunku 2.1 widoczny jest kod w języku Python służący do utworzenia takiego autoenkodera, który przeprowadzi analizę 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))
```

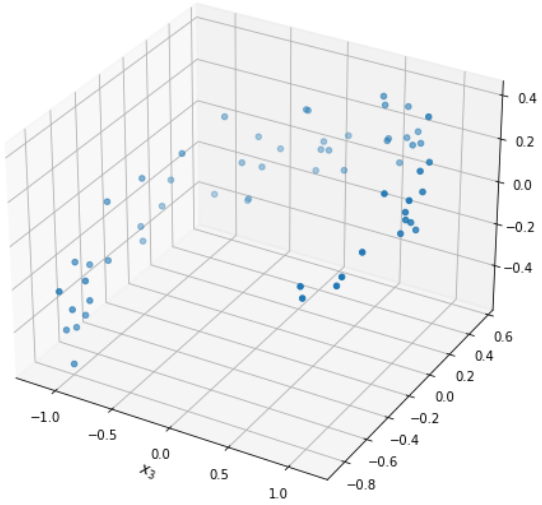
Rysunek 2.1: Tworzenie autoenkodera niedopełnionego przeprowadzającego PCA

Źródło: Opracowanie własne na podstawie [6]



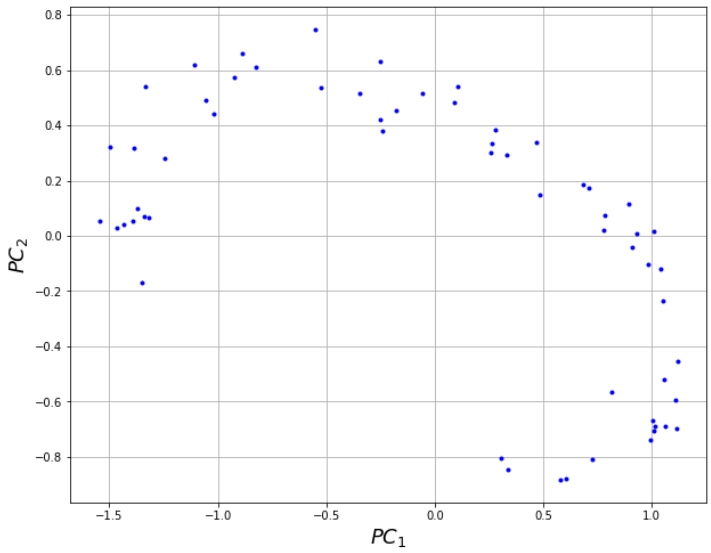
Rysunek 2.2: Architektura autoenkodera niedopełnionego
Źródło: [6]

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.



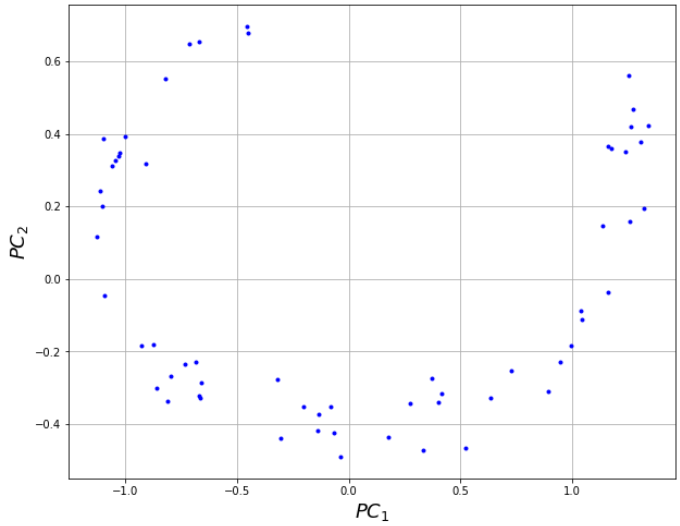
Rysunek 2.3: Wygenerowane dane trójwymiarowe

Źródło: Opracowanie własne na podstawie [6]

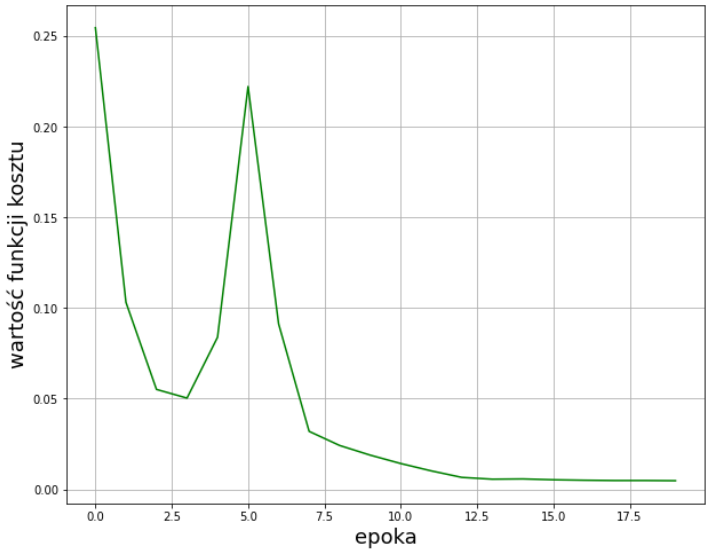


Rysunek 2.4: Rzutowanie dwuwymiarowe przy użyciu autoenkodera, zachowujące maksymalną wariancję

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

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

[10] 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