Zeszyty Naukowe WWSI, No 26, Vol. 16, 2022, pp. 21-60
DOI: 10.26348/znwwsi.26.21

Autoenkodery. Podstawy budowy wydajnych modeli uczenia maszynowego

Gabriel Rodewald*

Warszawska Wyższa Szkoła Informatyki

Streszczenie

Autoenkoder jest siecią neuronową złożoną z pary koder-dekoder. Koder odpowiada za redukcję wymiarowości danych w modelu przy jednoczesnym zachowaniu kluczowych cech, niezbędnych do odtworzenia danych wejściowych przez dekoder. Z uwagi na cechy architektury wewnętrznej wyodrębnia się autoenkodery deterministyczne oraz probabilistyczne. Istnieją wyspecjalizowane wersje autoenkoderów odpowiadające tematyce realizowanych modeli uczenia maszynowego, na przykład autoenkodery odszumiające, rekurencyjne, splotowe, wariacyjne lub rzadkie. W artykule zostały przedstawione jedynie najistotniejsze zagadnienia związane z autoenkoderami.

Słowa kluczowe — uczenie maszynowe, uczenie głębokie, autoenkodery, autoenkodery wariacyjne, autoenkodery odszumiające, autoenkodery rzadkie, autoenkodery konwolucyjne, autoenkodery rekurencyjne

1 Wstęp

Szukając rozwiązań problemów o dużej złożoności należy liczyć się ze wzrostem ilości przetwarzanych danych. Modele uczenia maszynowego powinny zatem być nie tylko precyzyjne, lecz także zoptymalizowane pod kątem wydajności. Przyjmując, że takim problemem byłoby wygenerowanie obrazu dowolnego obiektu, należałoby na wstępie wytypować najważniejsze cechy pierwowzoru obiektu. Powierzając to zadanie zespołowi złożonemu ze specjalistów, czas poświęcony na wyodrębnienie kluczowych cech obiektu i przeprowadzenie testów byłby nieopłacalny względem jakości uzyskanych wyników. Dzięki autoenkoderom proces dopasowania cech, które mają znaleźć się w modelu zostaje w pełni zautomatyzowany i podlega jedynie końcowej ocenie specjalisty. Poprawnie zaprojektowana sieć neuronowa nie dopuszcza do przeuczenia, redukuje wielowymiarowość danych oraz przeprowadza weryfikację cech w skończonym czasie.

*E-mail: gabrielrodewald@gmail.com

Zgłoszono do druku 14 stycznia 2022 r.

W artykule zostaną przedstawione wybrane zagadnienia dotyczące autoenkoderów. Jeżeli pojawi się taka potrzeba, będą także prezentowane listingi przedstawiające implementacje wybranych funkcji w języku *Python* (m.in. pochodzące z bibliotek Keras oraz PyTorch). W prezentowanych listingach pojawią się komentarze. Ponieważ w językach programowania nie używa się znaków diakrytycznych, to także nie używa się ich w komentarzach umieszczanych w programie. W związku z tym, że listingi będą prezentowane w formie rysunków przedstawiających zrzuty ekranu ze środowiska wspomagającego uruchamianie programów, dlatego komentarze znajdujące się wewnątrz prezentowanych listingów także pozbawione będą polskich znaków diakrytycznych.

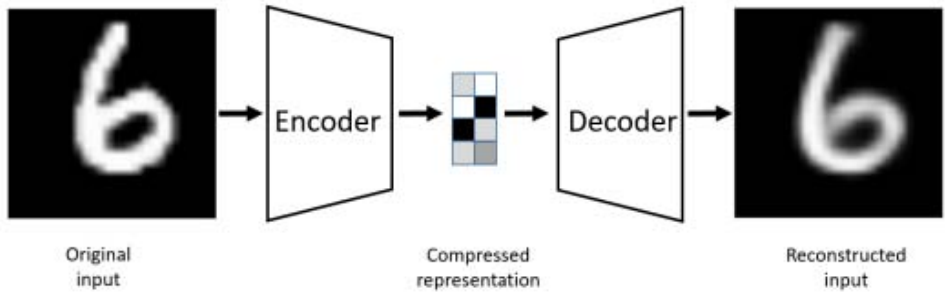
Niektóre pojęcia odnoszące się do autoenkoderów, funkcji strat, a także optymalizatorów, oznaczane są w artykule przy pomocy skrótów literowych. Aby ułatwić ich identyfikację, używane skróty zostaną tutaj wyjaśnione:

CAE	–	autoenkodery splotowe (ang. *Convolutional Autoencoders*),
CNN	–	konwolucyjne sieci neuronowe (ang. *Convolutional Neural Networks*),
DAE	–	autoenkodery odszumiające (ang. *Denoising Autoencoders*),
GAN	–	generatywne sieci przeciwstawne (ang. *Generative Adversarial Networks*),
GD	–	spadek wzdłuż gradientu (ang. *Gradient Descent*),
MAE	–	średni błąd bezwzględny (ang. *Mean Absolute Error*),
MSE	–	błąd średnio-kwadratowy (ang. *Mean Square Error*),
NLP	–	przetwarzanie języka naturalnego (ang. *Natural Language Processing*),
RMSE	–	średnia kwadratowa błędu, pierwiastek z MSE (ang. *Root Mean Square Error*),
RMSProp	–	propagacja średniokwadratowa (ang. *Root Mean Square Propagation*),
RNN	–	rekurencyjne sieci neuronowe (ang. *Recurrent Neural Networks*),
SAE	–	autoenkodery rzadkie zwane także rozproszonymi, (ang. *Sparse Autoencoders*),
SGD	–	stochastyczny spadek wzdłuż gradientu (ang. *Stochastic Gradient Descent*),
VAE	–	autoenkodery wariacyjne (ang. *Variational Autoencoders*).

2 Autoenkodery

Autoenkoder jest siecią neuronową złożoną z mniejszych, współpracujących ze sobą sieci neuronowych zwanych koderami oraz dekoderami. Przyjmuje się, że najprostszy autoenkoder jest zbudowany w oparciu o pojedynczą parę koder-dekoder [1]. Autoenkodery można dzielić według różnych kryteriów. Można je np. zaklasyfikować jako deterministyczne albo probabilistyczne. Według innych kryteriów można je podzielić na **autoenkodery niedopełnione** (ang. undercompleted autoencoders) oraz **autoenkodery przepełnione** (and. overcompleted autoencoders). Przy tym drugim podziale, w pierwszym przypadku liczba warstw ukrytych jest mniejsza niż liczba wejść, zaś w drugim przypadku liczba warstw ukrytych jest większa niż liczba wejść.

W pierwszym etapie koder redukuje dane wejściowe aby wyodrębnić kluczowe cechy reprezentujące obiekt. Dane, na przykład w postaci sekwencji pojedynczych obrazów, które zostały przekazane do kodera są mapowane na wektor i umieszczane w przestrzeni ukrytej. Sieci



Rysunek 1. Uogólniona architektura prostego autoenkodera [1]. Elementy od lewej to dane wejściowe, enkoder, warstwa ukryta o zmniejszonej liczbie danych, dekoder oraz odtworzone dane.

autoenkodera można trenować metodą end-to-end (przekazując dane w cyklu wejście-wyjście) lub przekazując każdą warstwę z osobna [1]. Na końcowym etapie dane otrzymuje dekoder w celu odtworzenia danych wejściowe z jak najmniejszą stratą (rys. 1)

Autoenkodery osadzając reprezentacje cech obiektu mogą operować na wielowymiarowych płaszczyznach. Mimo braku odgórnego limitu dotyczącego liczby warstw ukrytych zaleca się, aby nie było ich zbyt wiele. W przypadku nadmiernej liczby warstw pojawiają się dodatkowe problemy, takie jak powolne uczenie modelu, niewystarczające zasoby pamięci, pogarszająca się precyzja obliczeń. Problem ten jest opisany jako **przekleństwo wymiarowości**, czyli wzrost prawdopodobieństwa błędnych obliczeń z każdą dodatkową warstwą ukrytą modelu. Aurélien Géron w książce *Uczenie Maszynowe z użyciem Scikit-Learn i TensorFlow* [2] pisze, że dokonując losowego wyboru punktu na dwuwymiarowym kwadracie jednostkowym szansa na to, że punkt znajdzie się w odległości mniejszej niż 0,001 od brzegu wynosi zaledwie 0,4%. Dla przestrzeni tysiąc-wymiarowej szansa ta przekroczy 99,999999%. Przykład ten ilustruje jak może wrastać ryzyko błędnych obliczeń wraz ze zwiększaniem się liczby warstw ukrytych w modelu.

2.1 Kodery

(...) zadanie kodera polega na pobraniu obrazu wejściowego i zmapowanie go na punkt w przestrzeni ukrytej ([3], strona 71)[1]. Koder przyjmuje dane wejściowe (zazwyczaj wielowymiarowe), które następnie redukuje. Przekazuje na wyjście wektor o zmniejszonej liczbie wymiarów. Taki wektor nazywany jest *wektorem reprezentacji*, gdyż zawiera jedynie najistotniejsze partie danych wyselekcjonowane z całej populacji. Otrzymany wektor pozwala na odtworzenie pierwotnych danych.

W procesie przetwarzania danych koder używa warstw konwolucyjnych, inaczej nazywanych warstwami splotowymi. Warstwy konwolucyjne są obszarami zawierającymi filtry analizujące dane wejściowe w celu uwypuklenia pożądanych cech. Warstwy te poprzez użycie funkcji reduku-

[1]Chociaż cytat dotyczy przetwarzania obrazu, to w uogólnieniu dane będą pobierane z zestawu, a następnie mapowane na punkt w przestrzeni ukrytej.

jącej (w bibliotece Keras taką funkcją jest `Dense()`) zmniejszają wymiarowość i złączają dane z przestrzenią ukrytą. Przedstawiony tutaj opis jest dużym uproszczeniem, ponieważ architektura (funkcje aktywacji, liczba warstw ukrytych) kodera powinna być dobrana z uwzględnieniem specyfiki rozważanego problemu.

2.2 Dekodery

Dekoder przeprowadza dekompresję danych dążąc do ich jak najwierniejszego odtworzenia. W odróżnieniu od kodera używa konwolucyjnych warstw transponowanych. Najczęściej spotykaną architekturą dekodera jest odwrócony układ stosowanego w modelu kodera. Wiele bibliotek dedykowanych uczeniu maszynowemu wymaga, aby liczba wejść kodera i wyjść dekodera była taka sama.

Najpowszechniejszą praktyką jest jednoczesne szkolenie kodera i dekodera poprzez automatyczne zwracanie danych z wyjścia dekodera na wejście enkodera, co pozwala na wielokrotną selekcję cech. David Foster w książce *Deep learning i modelowanie generatywne* wskazuje, że biblioteka Keras posiada funkcje pozwalające zamknąć kompletny cykl w zaledwie trzech linijkach (pseudo) kodu ([3], strona 75):

```
model_input = encoder_input
model_output = decoder(encoder_output)
self.model = Model(model_input, model_output)
```

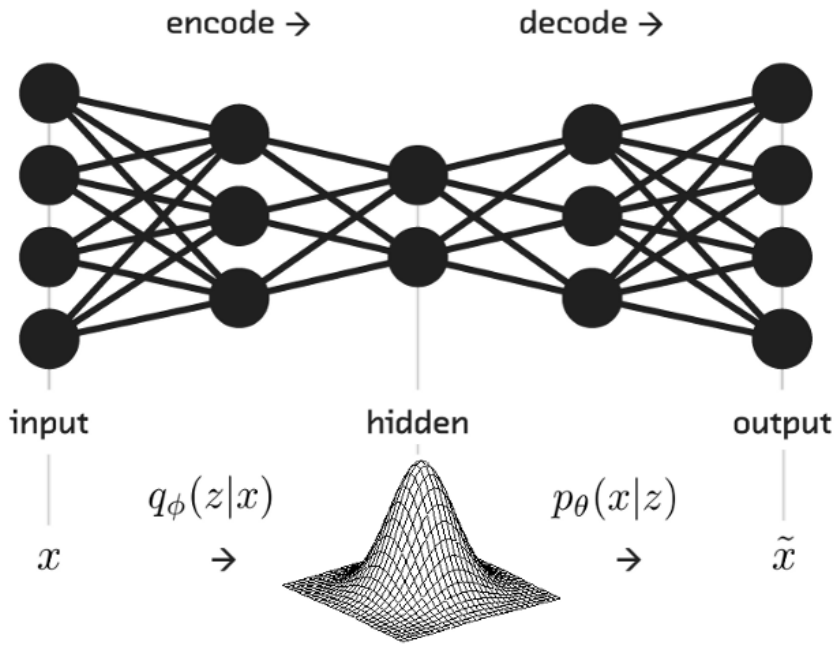
3 Rodzaje autoenkoderów

Autoenkodery różnią się między sobą pod względem architektury. Powodem różnic w architekturze są różnice w możliwych zastosowaniach, np. do detekcji wizerunku, rozróżniania obiektów, klonowania i modyfikacji dźwięku czy też do wykrywania nieautoryzowanych płatności. Przed przystąpieniem do tworzenia schematu autoenkodera należy wskazać cel, który ma zostać osiągnięty poprzez implementację modelu uczenia maszynowego (ang. *machine learning model*). Różnice pomiędzy architekturami modeli dotyczą najczęściej:

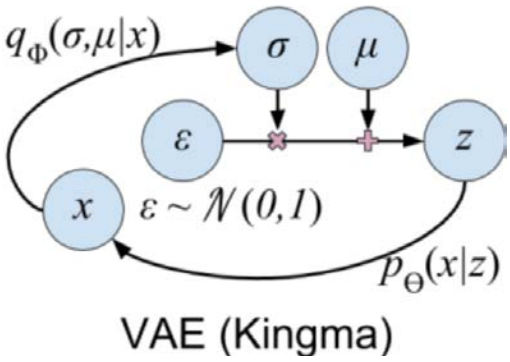
- liczby warstw w modelu,
- rodzaju, liczby i rozmieszczenia funkcji strat,
- optymalizatorów,
- funkcji aktywacji,
- wybranej technologii (biblioteki, frameworki, język programowania).

3.1 Autoenkodery wariacyjne

Autoenkodery wariacyjne (ang. *variational autoencoders*, *VAE*) stanowią odrębną, nową klasę autoenkoderów. Ze standardowymi autoenkoderami łączy je podobny schemat architektury wewnętrznej. VAE są probabilistyczne i mają potencjał generatywny. Wspomniana generatywność oznacza, że mogą tworzyć reprezentacje nowych obiektów bazując jedynie na ogólnych cechach

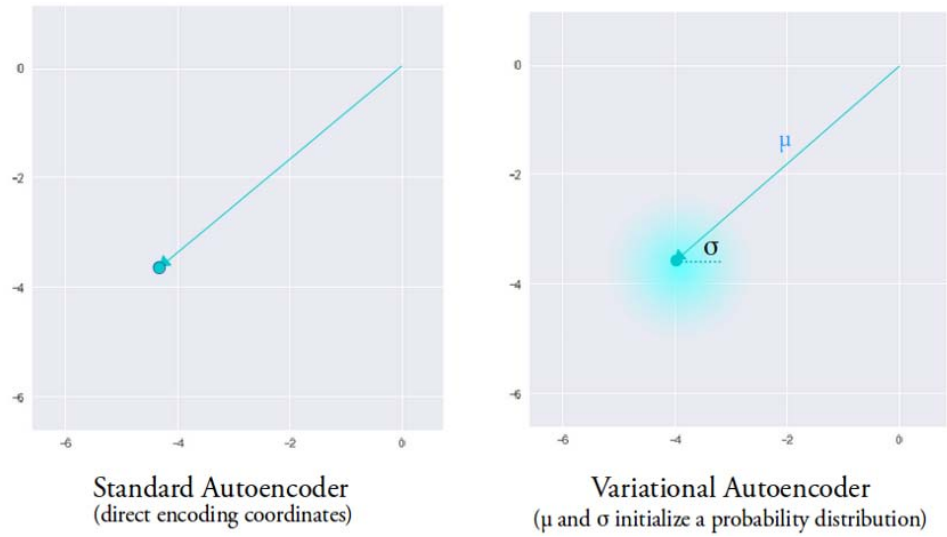


Rysunek 2. Struktura autoenkodera wariacyjnego [4].

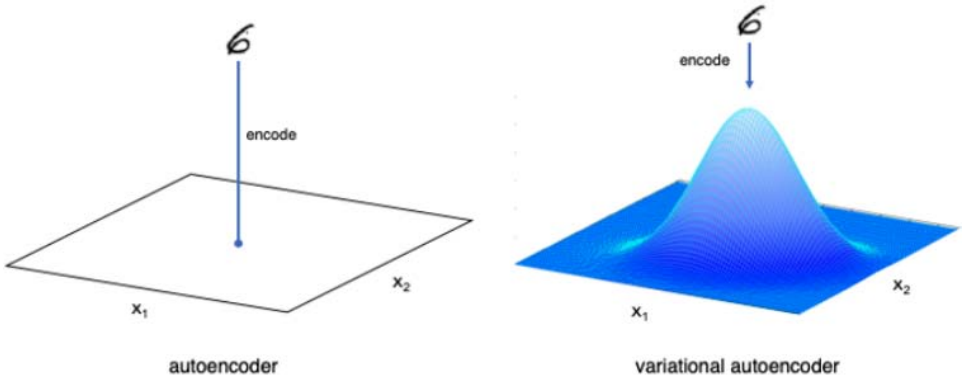


Rysunek 3. Model autokodowania w VAE. Strzałki na diagramie reprezentują mapowanie [5].

charakterystycznych egzemplarzy zaklasyfikowanych do podobnych grup [12]. „(...) *modelowanie dyksryminatywne próbuje oszacować prawdopodobieństwo, że obserwacja x należy do kategorii y. (...) model generatywny próbuje oszacować prawdopodobieństwo tego, że obserwacja*



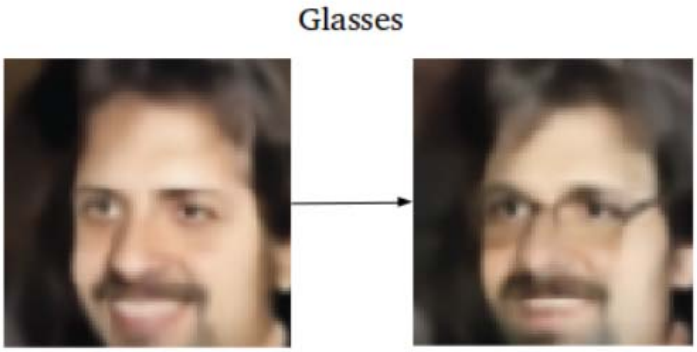
Rysunek 4. Sposób próbkowania danych w VAE [6].



Rysunek 5. Różnica pomiędzy zakodowaniem obiektu przez koder autoenkodera (z lewej) oraz przez koder autoenkodera wariacyjnego (z prawej) [3].

w ogóle się pojawi" ([3], str. 17).

Architektura VAE składa się z pary koder-dekoder, lecz to co ją wyróżnia to sposób próbkowania i przetwarzania danych w modelu (rys.: 2, 3, 4). Wartości przekazywane przez model są logarytmowane, dzięki czemu lepiej odzwierciedlają liczbę wejść i wyjść sieci neuronowej. Przekazany na wejście obraz zostaje zapisany do dwóch różnych wektorów: wektora μ reprezentującego średni punkt rozkładu oraz wektora wartości zlogarytmowanej dla każdego punktu. Z



Rysunek 6. Modyfikacja zdjęcia poprzez dodanie okularów [3].

Rysunek 7. Wygenerowane przez sieć GAN wizerunki ludzkich twarzy [7].

Rysunek 8. Wygenerowane przez sieć GAN wizerunki kotów[8].

takiego wielowymiarowego rozkładu normalnego próbkowane są wartości z w celu zakodowania obrazu. Służy do tego wzór:

$$z = \mu + \sigma\varepsilon, \tag{1}$$

Rysunek 9. Wygenerowane przez sieć GAN wizerunki koni [9].

Rysunek 10. Efekt przetransferowania stylu z obrazu na obraz [10].

gdzie:

z – punkt z rozkładu normalnego,
ε – punkt próbkowany z rozkładu normalnego,
σ – odchylenie standardowe,
μ – średnia.

Wartość ε pełni rolę strażnika gwarantującego umiejscowienie obiektów podobnej kategorii (zgrupowanymi wg wybranej, pożądanej cechy) w niewielkiej odległości od siebie. Dzięki temu dane poprawne nie zostaną błędnie sklasyfikowane jako wartości odstające.

Warto zauważyć, że następuje przejście od mapowania w dwóch wymiarach do mapowania wielowymiarowego (rys. 5). Autoenkodery wariacyjne są generatywne. Niekiedy nazywa się je także probabilistycznymi, ponieważ czynnik losowości przy generowaniu próbek nie zanika tak jak w standardowych autoenkoderach (część danych, które tworzą obiekt zawsze pochodzi z doboru losowego). Mogą samodzielnie tworzyć próbki imitując oraz jednocześnie modyfikując dane pochodzące z oryginalnego zestawu. W efekcie takiej transformacji na wyjściu otrzymywane są dane, które wcześniej nie występowały w zestawie. Zaprezentowane poniżej zdjęcie (rys. 6) jest przykładem *resyntezy obrazu* [13] polegającej na zmianie części obrazu poprzez nałożenie elementu, którego pierwotnie tam nie było.

Rysunek 11. Generowanie twarzy na podstawie głosu [11]. Pierwsza kolumna od lewej przedstawia docelowy wizerunek. Następna, od lewej, wskazuje uzyskany przez model obraz bazując na próbce głosowej. Kolejne pięć kolumn stanowi wybierane przez model możliwe wizerunki celem dopasowania głosu.

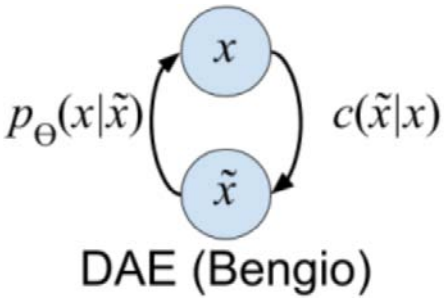
Dzięki rozwojowi modeli bazujących na VAE utworzono podobne sieci generatywne. Do najbardziej znanych zaliczają się **generatywne sieci przeciwstawne** (ang. *generative adversarial networks*, *GAN*), słynące z doskonałych rezultatów generowania (nieistniejących wcześniej) reprezentacji ludzkich twarzy (rys. 7) [14]. Możliwości generatywne tych sieci można zobaczyć na stronie internetowej [7]. Podobne generatory można znaleźć pod adresem [15].

Efektywność sieci nieco spada przy generowaniu wizerunków zwierząt, jak widać na grafice przedstawiającej koty (rys. 8). Ciekawostką jest, że największym problemem dla sieci GAN[2] jest utworzenie poprawnej reprezentacji konia widzianego z przodu (rys. 9).

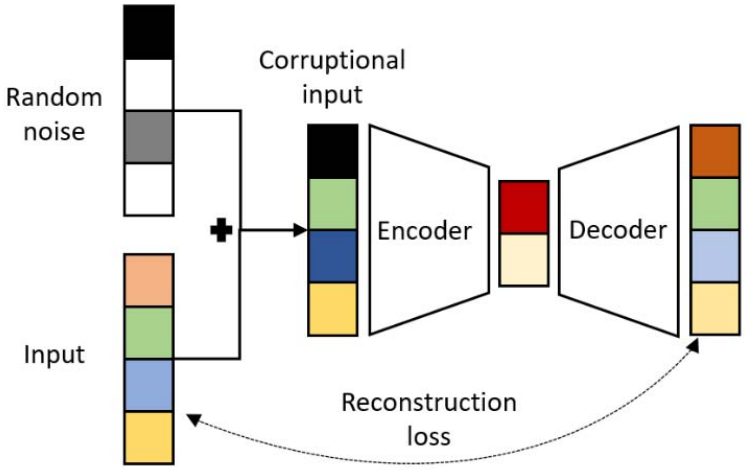
Z uwagi na rosnącą popularności zastosowań komercyjnych sieci GAN, tworzone są nowe, wyspecjalizowane autoenkodery, których podstawowym zadaniem jest manipulacja obrazem. Do tej kategorii można zaliczyć autoenkodery zajmujące się transferem stylu (rys. 10) (ang. *style transfer*, ST) [16]. Głowna idea działania ST opiera się na selekcji kluczowych cech jednego lub więcej obrazów i nałożeniu ich na obraz docelowy.

W abstrakcie pracy autorstwa H. Liang i in. [11] został przedstawiony zupełnie nowy koncept wykorzystania sieci GAN polegający na generowaniu ludzkich twarzy na podstawie nagranych próbek głosowych (rys. 11). Autorzy wskazują, że istnieje korelacja pomiędzy fizjonomią a cechami charakterystycznymi głosu.

[2]Informacja aktualna dla danych zebranych w styczniu 2022.



Rysunek 12. Struktura autoenkodera odszumiającego [1].

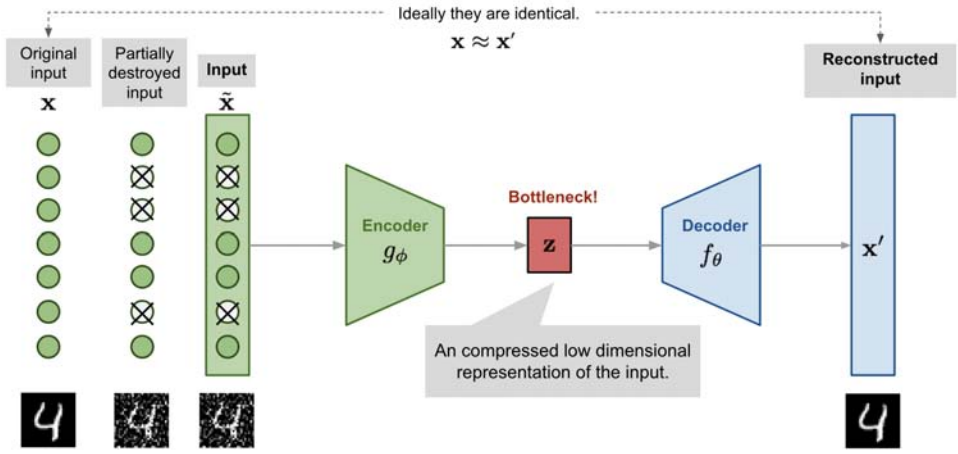


Rysunek 13. Próbkowanie danych w DAE [5].

3.2 Autoenkodery odszumiające

Autoenkoder odszumiający to rodzaj wielowarstwowej sieci perceptronowej złożonej z $2L - 1$ warstw ukrytych oraz $L - 1$ zestawów przypisanych wag [18]. Zadaniem autoenkoderów odszumiających (ang. *denoising autoencoders*, DAE) jest redukowanie szumu w danych. Schemat autoenkodera DAE został zaprezentowany na rysunku 12.

DAE najczęściej wykorzystywane są przy obróbce zdjęć, wideo oraz dźwięków. Model odwzorowujący dane wejściowe na wyjściowe w skali 1:1 (tzn. *X_wejściowe* = *X_wyjściowe*) nie jest korzystny, ponieważ nie jest w stanie odrzucić wartości oznaczonych jako szum. Chcąc zredukować taką możliwość wprowadza się do danych szum generowany w sposób stochastyczny. Mapowanie stochastyczne (wybór danych stanowiących szum) odbywa się według wzorów [17]:



Rysunek 14. Wizualizacja działania autoenkodera odszumiającego [17].

$$\widetilde{x}^{(i)} \sim \mathcal{M}_{\mathcal{D}}(\widetilde{x}^{(i)}|x^{(i)}), \tag{2}$$

gdzie:

$\mathcal{M}_{\mathcal{D}}$ – mapowanie danych oryginalnych na szum,
x – pojedyncza próbka danych ze zbioru, $x \in \mathcal{D}$,
$\widetilde{x}$ – zaszumiony odpowiednik próbki x,
$x^{(i)}$ – dane będące d-wymiarowymi wektorami dla których $x^{(i)=[x_1^{(i)},x_2^{(i)},\ldots,x_d^{(i)}]}$,

oraz:

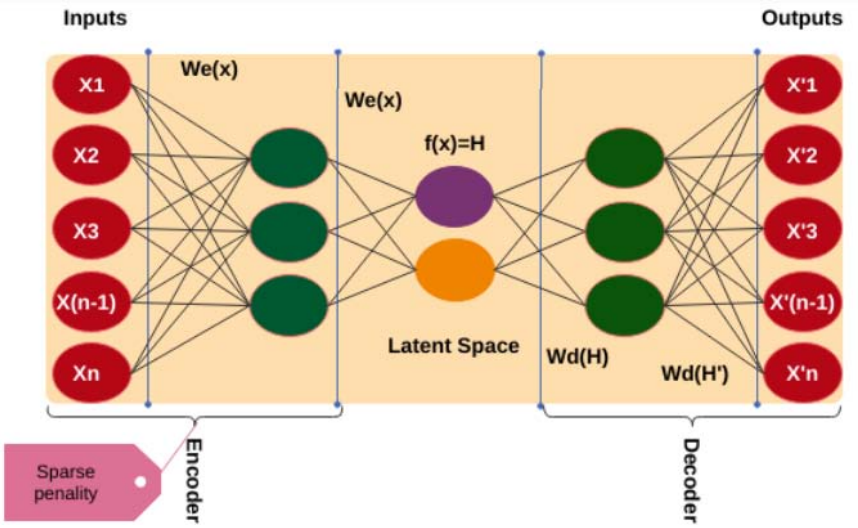
$$L_{DAE}(\theta, \phi) = \frac{1}{n}\sum_{i=1}^{n}(x^{(i)} - f_\theta(g_\phi(\widetilde{x}^{(i)})))^2, \tag{3}$$

gdzie:

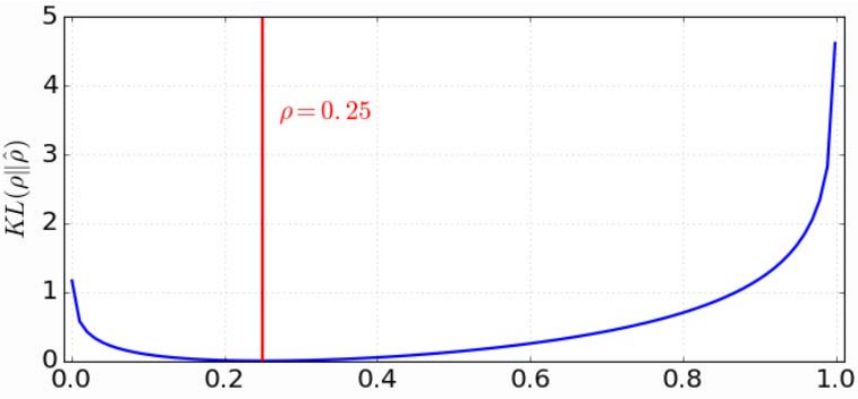
f_θ – funkcja dekodująca o parametrze θ,
g_ϕ – funkcja kodująca o parametrze ϕ.

Wprowadzenia szumu do danych można dokonać poprzez wprowadzenie nowych danych w postaci szumu do istniejącego zestawu, albo poprzez losowe zastąpienie szumem niektórych danych z zestawu (rys. 13). Rezultatem takiego działania będzie model potrafiący odtworzyć pierwotne dane bez szumu.

Chcąc zminimalizować redundancję opłacalne jest korzystanie z autoenkoderów niedopełnionych. Jak zaprezentowano na rysunku 14, dane początkowe są częściowo zastępowane szumem. Model powinien nauczyć się odróżniać istotne cechy od szumu i w konsekwencji odwzorować na wyjściu dane wejściowe bez szumu.



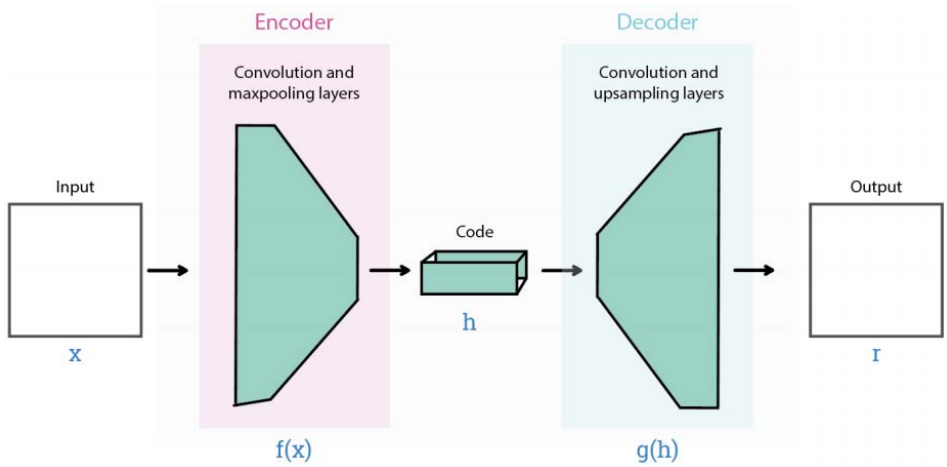
Rysunek 15. Budowa prostego autoenkodera rozproszonego [19].



Rysunek 16. Dywergencja Kullbacka-Leiblera pomiędzy rozkładami Bernoulliego dla $\rho = 0.25$ oraz $0 < \widehat{\rho} < 1$ [17].

3.3 Autoenkodery rzadkie

Autoenkodery rzadkie (ang. *sparse autoencoders*, SAE), zwane także rozproszonymi, pilnują by podczas procesu uczenia modelu nie przekroczyć żądanego procentu jednorazowo aktywnych w sieci neuronów. Jej przykładowa architektura została ukazana na rysunku 15. Wspomniany procent aktywnych neuronów nie może być ani zbyt mały (wówczas model nie będzie w stanie odtworzyć prawidłowo danych), ani zbyt duży (wówczas model będzie zwracał zbyt wiele cech



Rysunek 17. Podstawowa architektura autoenkodera splotowego[21].

redundantnych). Jego wartość jest monitorowana przez parametr straty rzadkości (ang. *sparsity loss*) zwany karą.

Funkcja kosztu (rys. 16) bazuje na **dywergencji Kullbacka-Leiblera** (ozn. KL) i dana jest następującym wzorem [20]:

$$\sum_{j=1}^{s_2} KL(\rho||\widehat{\rho}^j) = \rho \log \frac{\rho}{\widehat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\widehat{\rho}_j} \tag{4}$$

gdzie:

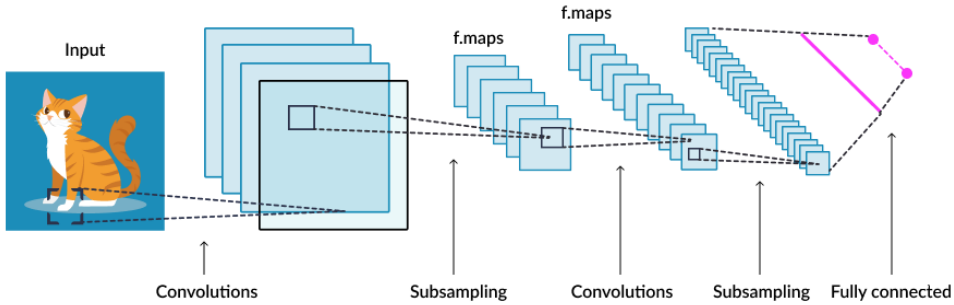
s_2 – liczba neuronów w warstwie ukrytej,
$||$ – rozbieżność (dywergencja) pomiędzy p a $\widehat{p}^j$,
$\rho, \widehat{\rho}_j$ – zmienne losowe rozkładu Bernoulliego, gdzie ρ jest docelowo pożądanym poziomem rozproszenia, a $\widehat{\rho}_j$ jest uśrednioną aktywacją jednostki ukrytej.

Im większa różnica pomiędzy ρ a $\widehat{\rho}_j$ tym większa będzie nałożona kara. Jeśli $\rho = \widehat{\rho}_j$, kara wyniesie 0. Uwzględniając powyższe, funkcja straty autoenkodera SAE wyraża się wzorem [20]:

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} KL(\rho||\widehat{\rho}^j), \tag{5}$$

gdzie:

W – waga (parametr) pomiędzy warstwą n a warstwą $n+1$,
b – błąd obciążenia (ang. *bias*) pomiędzy warstwą n a warstwą $n+1$,
β – parametr kontrolny kary rozproszenia (rzadkości).



Rysunek 18. Przykładowe pole recepcyjne [22].

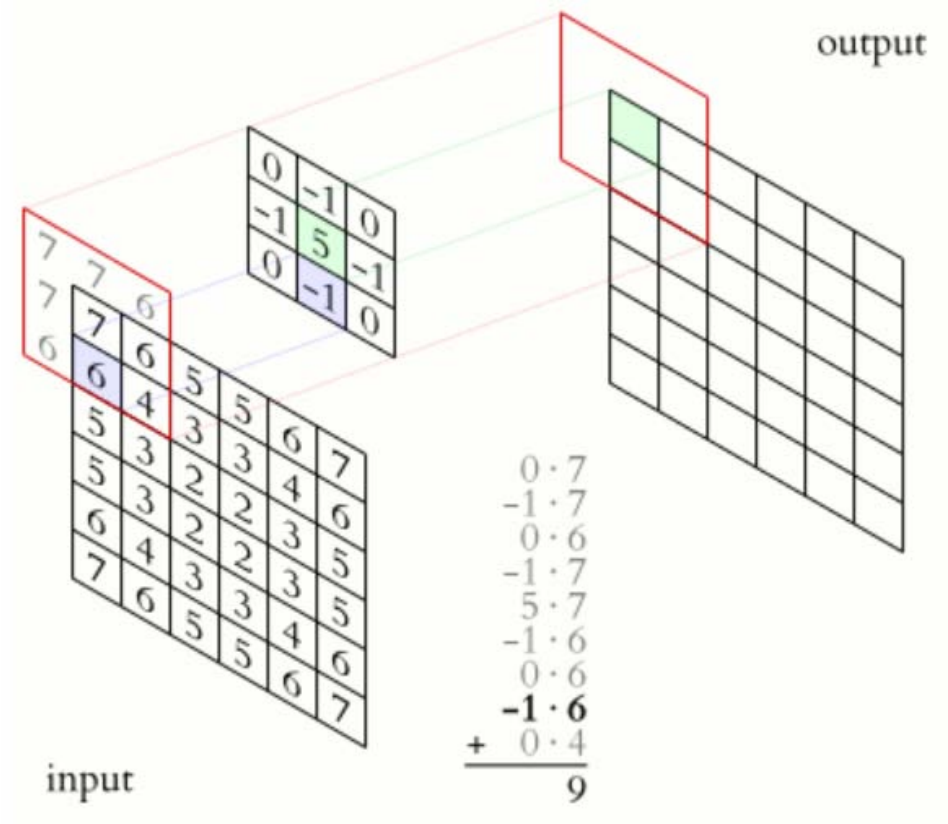
3.4 Autoenkodery konwolucyjne

Autoenkodery konwolucyjne, zwane także splotowymi, (ang. *convolutional autoencoders*, CAE) wykorzystują w swojej architekturze konwolucyjne sieci neuronowe (rys. 17) (ang. *convolutional neural networks*, CNN). CAE posiadają znaczną przewagę nad zwykłymi autoenkoderami w zakresie przetwarzania danych pochodzących z obrazów oraz wykrywania odległości pomiędzy osadzeniami słów w warstwach ukrytych przy rozkładach NLP [23].

Obraz cyfrowy złożony jest z pikseli mogących przybrać wartość liczbową od 0 do 255. W przeciwieństwie do innych typów danych, przy przetwarzaniu obrazów relacje pomiędzy pikselami są kluczowym czynnikiem pozwalającym na wierne odtworzenie reprezentacji przez dekoder. Głębokie sieci neuronowe dość dobrze radzą sobie z przetwarzaniem nieskomplikowanych obrazów (na przykład cyfr z bazy danych MNIST[3] [24]), lecz nie nadają się do przetwarzania obrazów o większej liczbie pikseli. Powodem jest zbyt duży przyrost połączeń neuronowych mających odwzorować obiekt w stosunku do opłacalności wyuczenia modelu (w tym przyrost czasowy jak i kosztowy, na przykład w postaci zużycia prądu, roboczogodzin pracowników nadzorujących proces uczenia modelu). W celu lepszego zrozumienia wydajności sieci CNN przy przetwarzaniu obrazu należy zwrócić uwagę na połączenia między warstwami. Pierwsza warstwa nie łączy się z każdym pikselem na obrazie (jak w przypadku standardowych sieci głębokich), a jedynie z okrojonym polem lokalnym (rys. 18) będącym wycinkiem obrazu na którym akurat skupia się algorytm. Warstwa pierwsza przekazuje tak okrojone dane do warstwy drugiej, która działa w taki sam sposób koncentrując się jedynie na małym lokalnym polu danych. Istotę problemu trafnie oddaje poniższy cytat ([3], strona 441): *„(...) obraz o rozmiarze 100x100 pikseli zawiera 10000 pikseli, a jeżeli pierwsza warstwa sieci składa się zaledwie z tysiąca neuronów (...), oznacza to łącznie 10 milionów połączeń, a to dotyczy zaledwie pierwszej warstwy. Problem ten zostaje rozwiązany w sieciach CNN poprzez wprowadzenie częściowo połączonych warstw i współdzielenia wag*".

Rozważając użycie sieci CNN należy zaznajomić się z kilkoma ważnymi pojęciami pozwa-

[3]Baza danych MNIST – baza danych odręcznych cyfr, używana do szkolenia różnych algorytmów uczenia maszynowego (ang. *MNIST database – Modified National Institute of Standards and Technology database*).



Rysunek 19. Wizualizacja działania jądra [25].

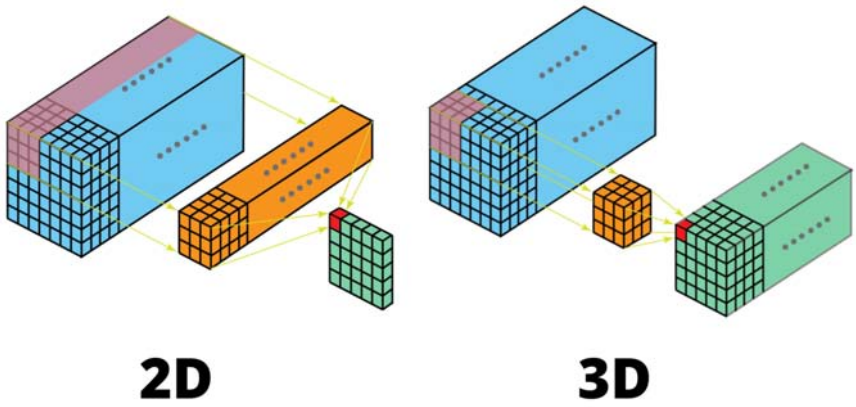
lających dogłębniej zrozumieć ich działanie. Są to: jądro splotowe (ang. *convolutional kernel*), filtr oraz pooling.

3.4.1 Jądro splotowe

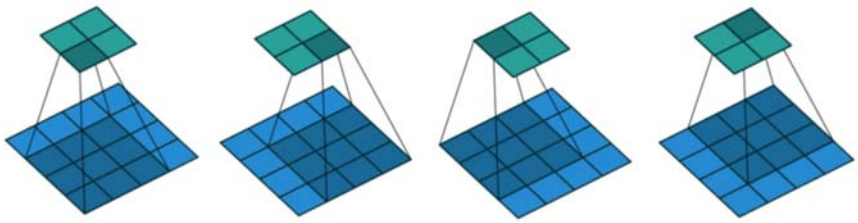
Jądro splotowe (ang. *convolutional kernel*) to zestaw danych określany przez zadaną wielkość macierzy objęty *polem recepcyjnym* (rys. 18). Poprzez pole recepcyjne należy rozumieć obszar będącym wycinkiem całości obrazu, na którym aktualnie skupia się algorytm. Jądro jest macierzą wag, w której każda z wag mnożona jest przez dane wejściowe (rys. 19).

3.4.2 Filtry

Filtry to zestawy jąder wyznaczone do mapowania danych obrazu. Odpowiadają za przekazywanie wag i pamiętanie relacji przestrzennych pomiędzy pikselami. Do obrazów czarno-białych



Rysunek 20. Porównanie filtrów 2D i 3D [25].



Rysunek 21. Proces mapowania danych przez filtr [25].

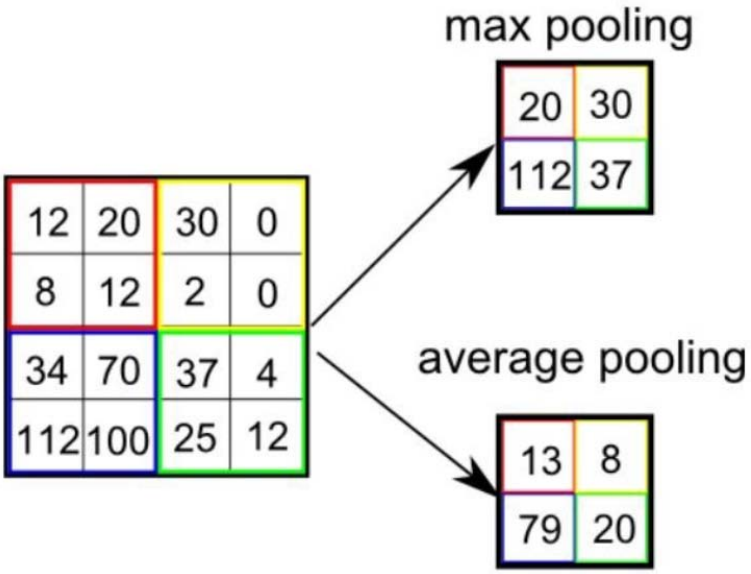
stosuje się przeważnie filtry pionowe i poziome, zaś do kolorowych filtry bazujące na kanałach RBG (ang. *Red*, *Blue*, *Green*).

Wymiar budowy pojedynczego filtra to *n+1*, gdzie *n* to liczba jąder, zaś 1 jest dodatkowym wymiarem narzucanym odgórnie. Jeśli suma jąder stanowi macierz 2D, to filtr będzie odpowiednikiem 3D (rys. 20). Każde z jąder odpowiada za przetwarzanie danych przechwyconych na dedykowanym kanale wejściowym. Filtr może przemieszczać się w sposób domyślny piksel po pikselu lub mieć określony **krok** (ang. *stride*), który wykonuje poruszając się po obrazie (rys. 21).

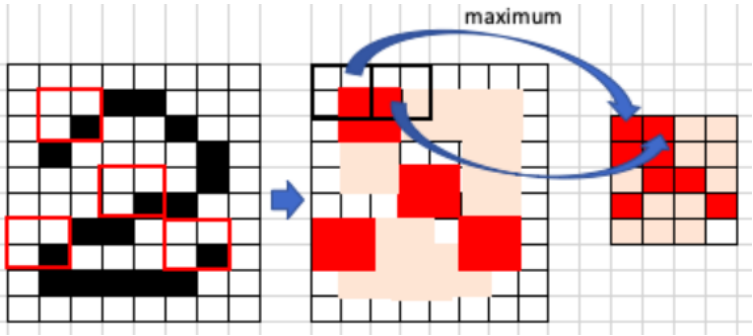
W pierwszej iteracji wagi zebrane przez filtry szkolące sieć CNN są losowe. Przy kolejnych iteracjach filtry będą coraz lepiej dopasowane dzięki wyodrębnieniu cech charakterystycznych obrazów. Sieć będzie mogła odtwarzać obraz ze znacznie bardziej zredukowanej macierzy badanego obiektu.

3.4.3 Pooling

Pooling jest procesem redukcji (poprzez łączenie, scalanie) danych zbliżonych pod względem wartości znajdujących się obok siebie [28]. Ponieważ na obrazach piksele pozostające w bliskim



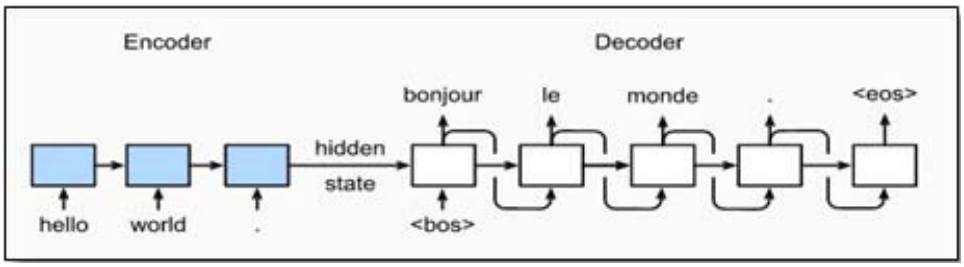
Rysunek 22. Pooling (łączenie, gromadzenie) – zmniejszanie obrazu poprzez scalanie podobnych wartości sąsiadujących [26].



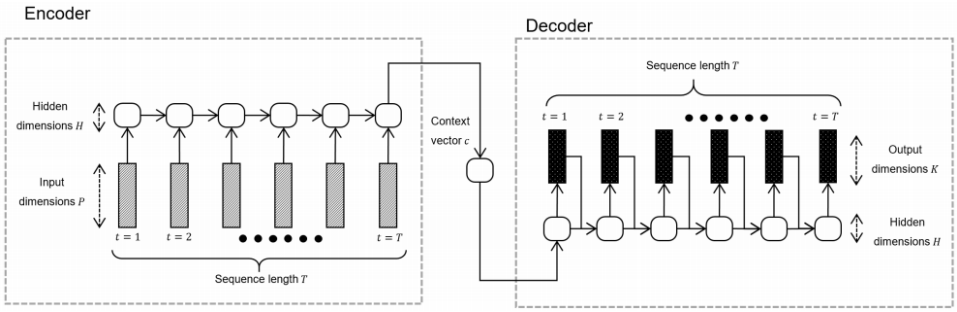
Rysunek 23. Schemat wyboru danych przy zastosowaniu warstwy max pooling [27].

sąsiedztwie mają zazwyczaj podobny kolor (oznaczony zbliżoną wartością liczbową), można założyć, że istnieje taki moment do którego opłacalne jest redukowanie sąsiadujących podobnych wartości.

Spośród najczęściej stosowanych metod wyróżniamy **max pooling** (rys. 22), gdzie warstwa zakoduje tylko największe wartości wskazane przez dane okno macierzy oraz **average pooling**, gdzie wybrane zostaną wartości uśrednione. Na rysunku 23 pokazano wyselekcjonowane wartości dla obu przypadków.



Rysunek 24. Architektura sieci RNN na przykładzie NLP (przekład z języka angielskiego na język francuski) [29]. Rysunek przedstawia brak równości w sekwencjach wejście-wyjście.



Rysunek 25. Architektura sieci RNN. Koder oraz dekoder są wielowarstwowymi sieciami RNN [30].

3.5 Autoenkodery rekurencyjne

Autoenkoder każdorazowo powinien być oceniany pod względem wydajności w oparciu o typ wprowadzanych danych. Przykładowo, przetwarzając dane sekwencyjne można posłużyć się siecią CNN, choć dużo lepszym wyborem będzie zastosowanie rekurencyjnych sieci neuronowych (ang. *recurrent neural networks*, RNN).

Sieci RNN są wykorzystywane w nauczaniu nienadzorowanym oraz wszędzie tam, gdzie przetwarza się język naturalny (ang. *natural language processing*, *NLP*), szeregi czasowe (ang. *time series*) lub obrazy (jeśli spełniony zostanie warunek przetwarzania sekwencyjnego danych wejściowych). Przykładowy schemat budowy został przedstawiony na rysunku 24). W sieciach RNN koder może otrzymać dane o innej długości niż te, które trafią do dekodera (rys. 25). Jest to istotne zwłaszcza przy kodowaniu i dekodowaniu języka naturalnego, ponieważ struktury języków przeważnie są od siebie różne. Przykładowo, trzy słowa w zdaniu w języku polskim *„Słońce jest żółte."* po przetłumaczeniu na język angielski staną się czterema słowami – *„The sun is yellow."*. Otrzymane na wejściu dane w wektorze a o długości n i wymiarowości m na etapie kodera są mapowane na wektor b, będący reprezentacją umożliwiającą dekoderowi odtworzenie

danych, i w takim stanie są przekazane do dekodera.

4 Problemy związane z zastosowaniem autoenkoderów

Większość autoenkoderów została stworzona w celu optymalizacji procesu rozwiązywania problemów ściśle określonej kategorii. Niedopasowanie rodzaju autoenkodera do kategorii problematyki rzutuje na jakość predykcji lub generatywności modelu. W celu lepszego zrozumienia zagadnienia przyjmuje się, że omawiane poniżej problemy dotyczą danych, które są:

1. sekwencją obrazów złożoną z pojedynczych cyfr,
2. cyframi napisanymi odręcznie, a następnie zeskanowanymi,
3. obrazami dwuwymiarowymi, w odcieniach w skali szarości (reprezentowanymi przez rozłożenie odcieni szarości pikseli na osiach x oraz y).

W konsekwencji rozważane są dwa następujące problemy:

Problem 1: Brak przeskalowanego rozłożenia reprezentacji grup.
Autoenkoder nie daje gwarancji poprawnego osadzenia cyfr w przestrzeni ukrytej. Reprezentacje podobnych obiektów nie muszą być równomiernie rozłożone względem punktów $(0, 0)$ na osiach x, y.

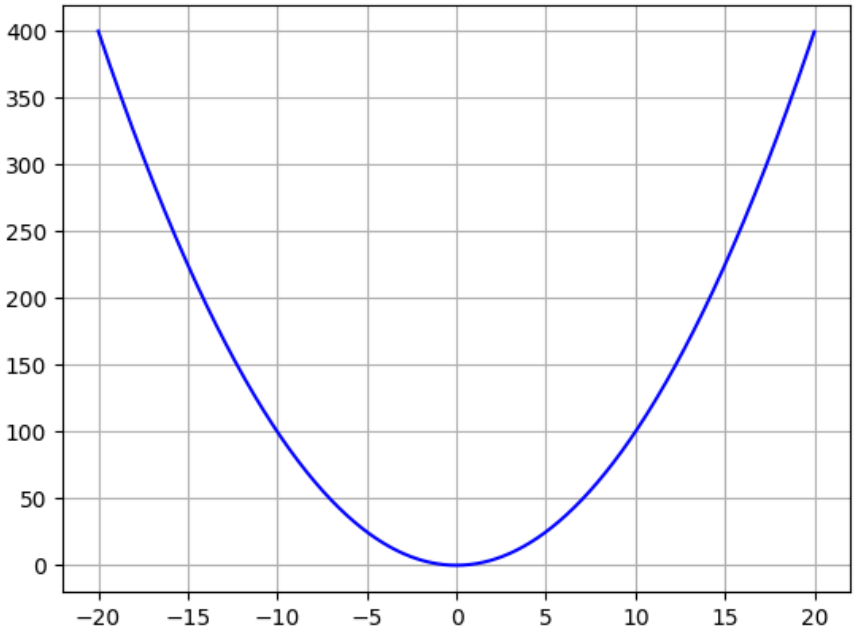
Problem 2: Niewłaściwe proporcje wielkości grup.
Przyjmując, że cyfra 0 występuje w zbiorze 50 razy, a cyfra 7 występuje 5 razy, można oczekiwać, że osadzenie grupy zajmie więcej miejsca dla cyfr 0 niż 7. Jak zauważa D. Foster [3], autoenkodery nie zapewniają *ciągłości przestrzeni* i cyfry rozłożone blisko siebie na osiach nie muszą przynależeć do tej samej grupy. Wpływa to na błędne typowanie wyniku przy losowym wyborze cyfry i odtwarzaniu oryginalnego jej obrazu.

Problemy z poprawną klasyfikacją danych pojawiają się już przy dwóch wymiarach przestrzeni ukrytej. Autoenkodery o prostej architekturze są nieefektywne przy bardziej zaawansowanych projektach, w których dane są osadzane w *n*-wymiarach. Rozwiązaniem są autoenkodery wariacyjne, które zostały opisane w podrozdziale o autoenkoderach wariacyjnych (p. 3.1).

4.1 Funkcje strat

Funkcja strat (zwana również *funkcją kosztu* lub *kosztem*) to informacja zwrotna w postaci obliczeń wskazująca jak daleko od pożądanego efektu znajduje się przetworzona próbka danych otrzymana na wyjściu. Parafrazując, na podstawie wyliczonego błędu wiadomo, czy autoenkoder zmierza w prawidłowym kierunku poszukując jak najlepszego odwzorowania oryginalnych danych wejściowych [31].

Funkcje strat opisuje się na wiele sposobów. Poniżej zostały wyszczególnione najczęściej używane funkcje strat zaimplementowane w bibliotekach Keras i PyTorch. Tam gdzie było to możliwe wskazano alternatywne rozwiązanie bez zastosowania bibliotek.



Rysunek 26. Wizualizacja straty MSE [31]

```
#klasa straty MSE
tf.keras.losses.MeanSquaredError(
reduction = "auto",
name = "mean_squared_error"
)

# przekazanie wartosci wejsciowej oraz predykowanej
y_true = [[0., 1.], [0., 0.]]
y_pred = [[1., 1.], [1., 0.]]

# zmienna mse, do ktorej przekazujemy dane
mse = tf.keras.losses.MeanSquaredError()
mse(y_true, y_pred).numpy()

# wywolanie funkcji z wagami
mse(y_true, y_pred, sample_weight = [0.7, 0.3]).numpy()
```

Rysunek 27. Strata MSE. Implementacja przy pomocy biblioteki Keras [32]

```
import torch
import torch.nn as nn

input = torch.randn(3, 5, requires_grad = True)
target = torch.randn(3, 5)
mse_loss = nn.MSELoss()
output = mse_loss(input, target)
```

Rysunek 28. Strata MSE. Implementacja przy pomocy biblioteki PyTorch [33]

```
import numpy as np

def mse_loss(y_pred, y_true):
squared_error = (y_pred - y_true) ** 2
sum_squared_error = np.sum(squared_error)
loss = sum_squared_error / y_true.size
return loss
```

Rysunek 29. Strata MSE. Implementacja bez użycia bibliotek uczenia głębokiego [31]

4.1.1 MSE – błąd średnio-kwadratowy

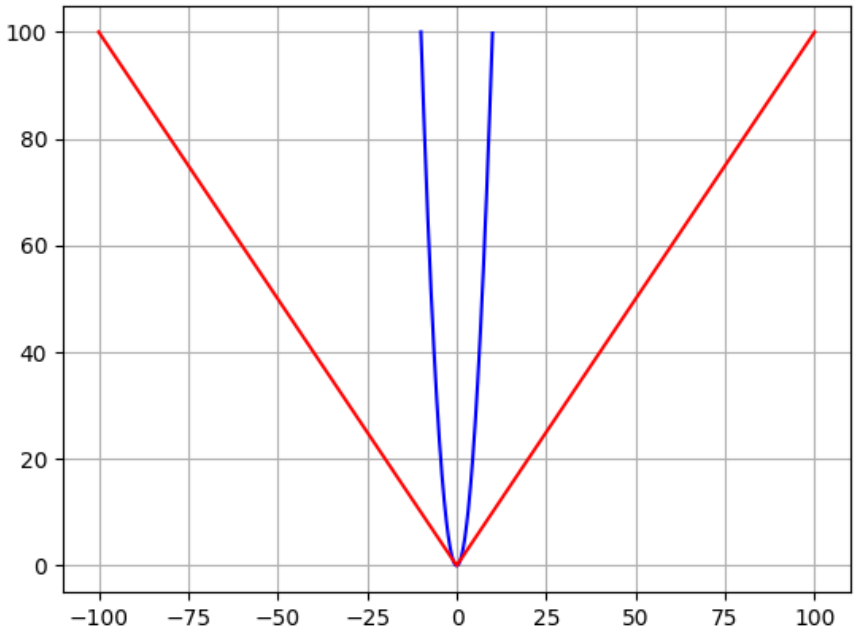
Jednym ze sposobów opisu funkcji strat jest opis w postaci błędu średnio-kwadratowego (rys. 26). Błąd średnio-kwadratowy (czasem średni błąd kwadratowy) oznacza się skrótem MSE (ang. *mean square error*) [31]. Dla każdej próbkowanej wartości obliczany jest kwadrat różnicy między wyjściem rzeczywistym a oczekiwanym. Wyniki otrzymane dla próbkowanych wartości są uśredniane. Przyjmuje się, że im mniejszy współczynnik MSE tym lepszy jest model. MSE dany jest wzorem:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2, \tag{6}$$

gdzie:

y_i – rzeczywista wartość,
$\widehat{y}_i$ – prognozowana wartość,
$i = 1, 2, 3, \ldots n$.

Na rysunkach 27 oraz 28 przedstawiono przykłady obliczania funkcji strat MSE przy pomocy bibliotek Keras oraz PyTorch. Na rysunku 29 ukazano zastosowanie funkcji starty MSE bez użycia bibliotek głębokiego uczenia maszynowego.



Rysunek 30. Porównanie funkcji strat MAE (kolor czerwony) z funkcją straty MSE (kolor niebieski) [31].

```
import torch
import torch.nn as nn
input = torch.randn(3, 5, requires_grad = True)
target = torch.randn(3, 5)
mae_loss = nn.L1Loss()
output = mae_loss(input, target)
```

Rysunek 31. Strata MAE. Implementacja przy pomocy biblioteki Keras [32]

4.1.2 RMSE – średnia kwadratowa błędów

Inną możliwością opisu funkcji straty jest średnia kwadratowa błędu, RMSE (ang. *root mean square error*), będąca pierwiastkiem zdefiniowanego uprzednio błędu średnio kwadratowego MSE [31]:

$$RMSE = \sqrt{MSE}. \tag{7}$$

```
import numpy as np

def mae_loss(y_pred, y_true):
abs_error = np.abs(y_pred - y_true)
sum_abs_error = np.sum(abs_error)
loss = sum_abs_error / y_true.size
return loss
```

Rysunek 32. Strata MAE. Implementacja przy pomocy biblioteki PyTorch [33].

```
import numpy as np

def mae_loss(y_pred, y_true):
abs_error = np.abs(y_pred - y_true)
sum_abs_error = np.sum(abs_error)
loss = sum_abs_error / y_true.size
return loss
```

Rysunek 33. Strata MAE. Implementacja bez użycia bibliotek uczenia głębokiego [31].

4.1.3 MAE – średni błąd bezwzględny

Funkcja straty może być zaprezentowana również jako średni błąd bezwzględny, MAE (ang. *mean absolute error* (rys. 30) [31]. W tym przypadku uśredniany jest moduł różnicy pomiędzy wartością rzeczywistą a prognozowaną:

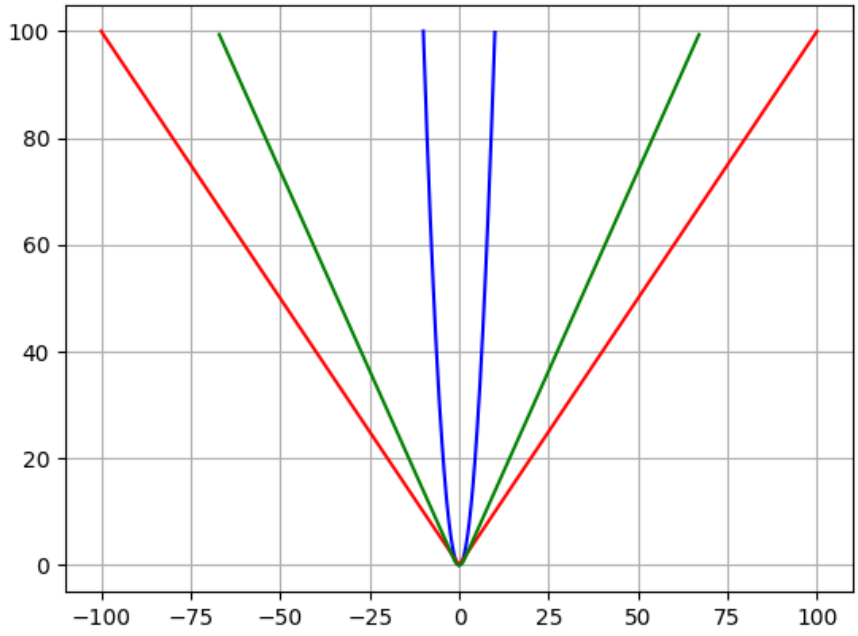
$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y}_i|, \tag{8}$$

gdzie:

y_i – rzeczywista wartość,
$\widehat{y}_i$ – prognozowana wartość,
$i = 1, 2, 3, \ldots n$.

Funkcja MAE nie koryguje wag w przypadku występujących wartości odstających (ang. *outlier*). Wagi pozostają w zależności liniowej względem punktów, zatem położenie wartości odstających nie są korygowane (w przeciwieństwie do wyniku MSE). Jeśli w założeniu projektu można pozwolić na odrzucenie wartości odstających, funkcja MAE będzie lepszym wyborem niż MSE.

MAE można zaimplementować przy pomocy bibliotek Keras i PyTorch, jak również bez użycia bibliotek. Przykłady poszczególnych implementacji funkcji straty MAE obrazują następujące algorytmy: biblioteka Keras (rys. 31), biblioteka PyTorch (rys. 32), bez użycia bibliotek głębokiego uczenia (rys. 33).



Rysunek 34. Porównanie straty Hubera z funkcjami strat MSE oraz MAE [31]: kolor niebieski – funkcja MSE, kolor czerwony – funkcja MAE, kolor zielony – strata Hubera.

```
# klasa straty Huber
tf.keras.losses.Huber(delta = 1.0, reduction="auto", name="
                                    huber_loss")

# przekazanie wartosci wejsciowej oraz predykowanej
y_true = [[0, 1], [0, 0]]
y_pred = [[0.6, 0.4], [0.4, 0.6]]

# zmienna h, do ktorej przekazujemy dane
h = tf.keras.losses.Huber()
h(y_true, y_pred).numpy()

# wywolanie funkcji z wagami
h(y_true, y_pred, sample_weight = [1, 0]).numpy()
```

Rysunek 35. Strata Hubera. Implementacja przy pomocy biblioteki Keras [32].

```
import torch
import torch.nn as nn
torch.nn.HuberLoss(reduction='mean', delta=1.0)
```

Rysunek 36. Strata Hubera. Implementacja przy pomocy biblioteki PyTorch [33].

```
import numpy as np

def huber_loss(y_pred, y, delta = 1.0):
huber_mse = 0.5 * (y - y_pred)** 2
huber_mae = delta * (np.abs(y - y_pred) - 0.5 * delta)
return np.where(np.abs(y - y_pred) <= delta, huber_mse, huber_mae)
```

Rysunek 37. Strata Hubera. Implementacja bez użycia bibliotek uczenia głębokiego [31].

4.1.4 Strata Hubera

Przy założeniu, że posiadany zbiór danych nie został zweryfikowany, nie jest znany procent wartości odstających. Wybierając MAE lub MSE jako funkcję straty należy uwzględnić pewien procent zafałszowanych danych, choćby z powodu nadmiernego uśrednienia wag lub całkowitego odrzucenie wartości odstających. Problem może być rozwiązany przy użyciu funkcja straty Hubera (rys. 34), będącej połączeniem MSE oraz MAE [31]. Zgodnie ze wzorem poniżej, dla wartości mniejszych lub równych δ strata Hubera uruchomi funkcję straty MSE, a dla wartości większych aktywuje MAE:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{gdy} \quad |y - f(x)| \leqslant \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{gdy} \quad |y - f(x)| > \delta, \end{cases} \tag{9}$$

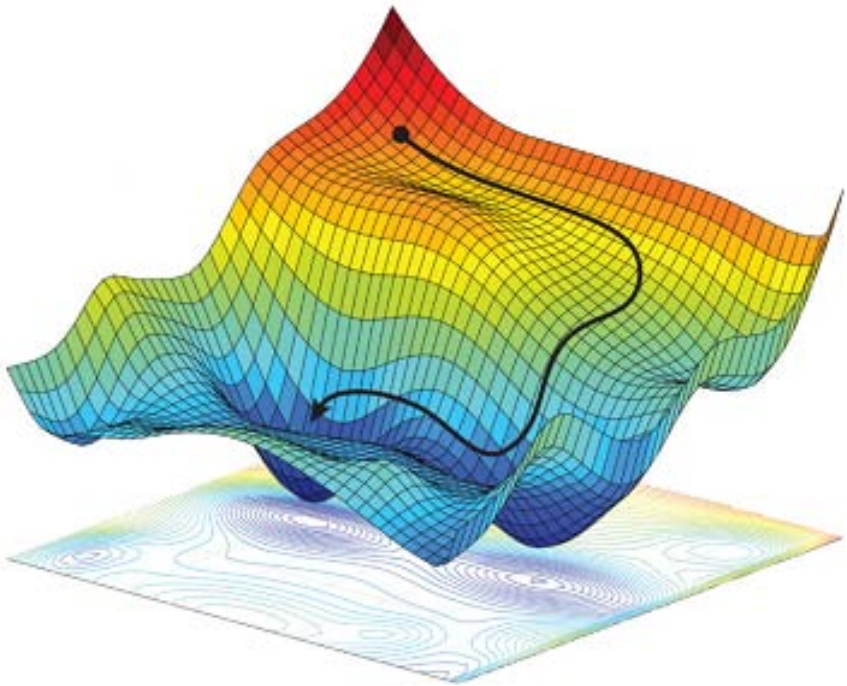
gdzie:

δ – parametr stały funkcji L,
$y, f(x)$ – argumenty funkcji L.

Stratę Hubera można zaimplementować przy użyciu bibliotek Keras i PyTorch lub także bez wspomagania się bibliotekami. Przykłady poszczególnych implementacji: biblioteka Keras (rys. 35), biblioteka PyTorch (rys. 36), bez użycia bibliotek uczenia głębokiego (rys. 37).

4.2 Algorytmy optymalizacyjne

Algorytmy optymalizacyjne, zwane potocznie optymalizatorami, są jednym z czynników wpływających na zmniejszenie czasu uczenia modelu sieci głębokiej [36]. Ich zadanie polega na przyspieszeniu pracy modelu, tak aby w jak najkrótszym czasie osiągnąć globalne minimum (rys.

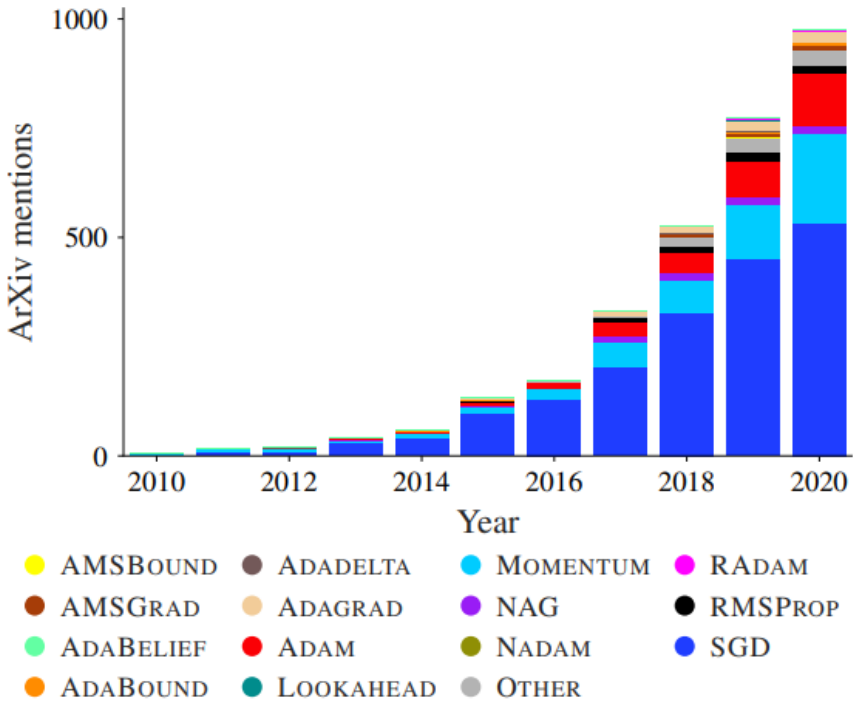


Rysunek 38. Wizualizacja poszukiwania globalnego minimum przez algorytm optymalizacyjny [34].

38). Aurélien Géron wskazuje na pozostałe trzy aspekty mające duży wpływ na skrócenie czasu uczenia [2]:

- dobór odpowiedniej funkcji aktywacji,
- prawidłowa inicjalizacja wag,
- użycie normalizacji wsadowej.

Dostrajanie optymalizatorów odbywa się poprzez zmianę ich hiperparametrów. Z uwagi na dużą liczbę potencjalnych kombinacji, decyzję o ich wyborze należy dokładnie przemyśleć w oparciu o wymogi techniczne projektu. Jak podają autorzy pracy *Descending through a Crowded Valley — Benchmarking Deep Learning Optimizers* [35] na rok 2020 szacowano ponad 100 optymalizatorów dedykowanych modelowaniu sieci w uczeniu głębokim (rys. 39). Niżej zostanie zaprezentowane zestawienie kilku optymalizatorów, najczęściej wykorzystywanych w projektach uczenia głębokiego



Rysunek 39. Liczba odniesień w publikacjach w ujęciu rocznym na portalu arXiv (bazując na tytułach oraz abstraktach) na temat danego optymalizatora [35].

4.2.1 GD oraz SGD

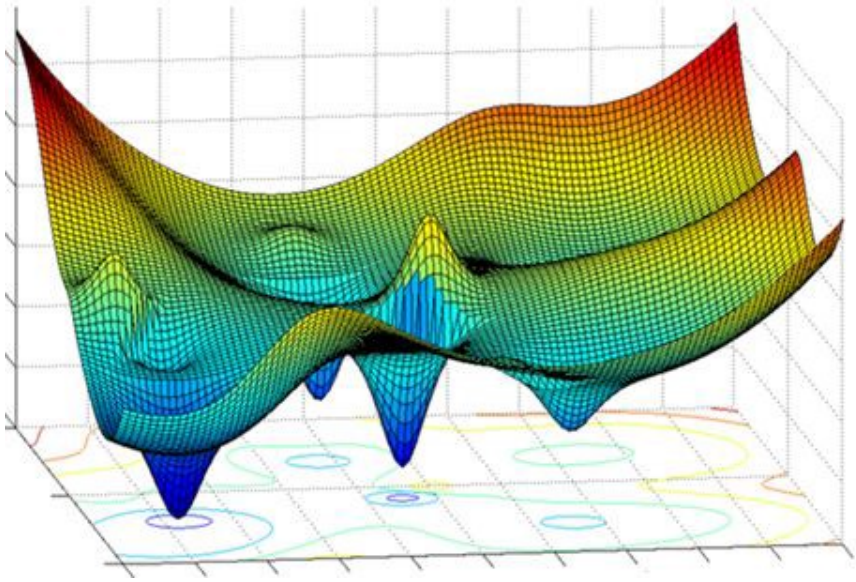
Spadek wzdłuż gradientu (ang. *gradient descent*, GD) oznacza nadzorowanie wag i ich korygowanie w celu osiągnięcia minimum [37]. Algorytm sprawdza aktualną pierwszą pochodną funkcji straty i na jej podstawie dokonuje decyzji o dokonaniu minimalizacji. Dzięki **propagacji wstecz** (ang. *back propagation*) warstwy sieci przekazują między sobą aktualną wartość straty. Zdarza się, że dalsze obliczenia zostają wstrzymane z powodu ugrzęźnięcia algorytmu w lokalnym minimum (rys. 40).

Z dostępnych n danych w zestawie, algorytm każdorazowo wybiera wszystkie n elementów. Z tego powodu zaleca się, aby zbiór danych był niewielki, ponieważ wagi ulegają skorygowaniu dopiero po przeliczeniu danych dla pełnego zestawu wejściowego. W przeciwnym razie czas potrzebny na ukończenie obliczeń może być nieopłacalny względem wydajności. Algorytm wykorzystuje następującą formułę:

$$\theta_0 = \theta - \alpha \bigtriangledown_\theta J(\theta), \tag{10}$$

gdzie:

θ_0 – nowe przybliżenie rozwiązania,



Rysunek 40. Przykład globalnego minimum i lokalnych minimów [37]. Lokalnych minimum może być wiele.

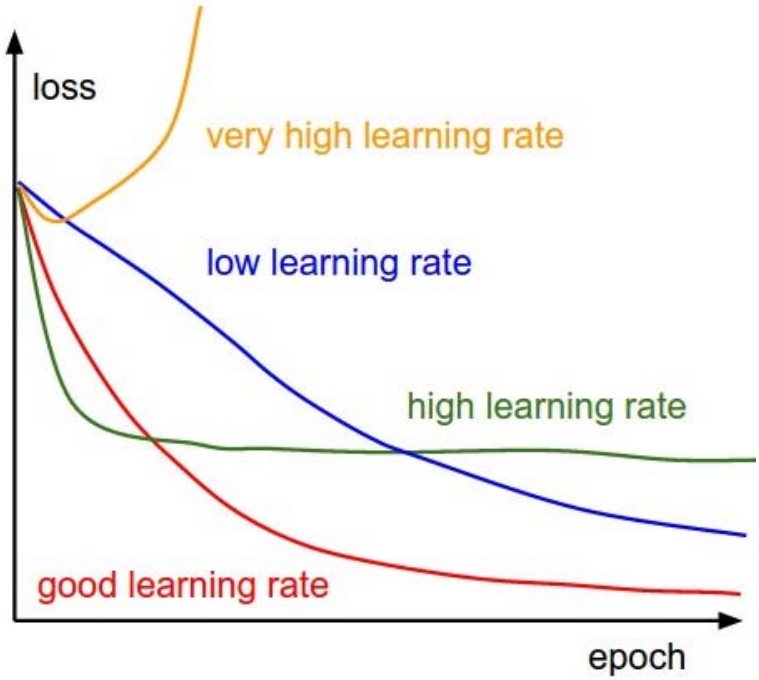
θ – aktualne przybliżenie rozwiązania,
α – współczynnik określający tempo uczenia się algorytmu,
$\bigtriangledown_\theta J(\theta)$ – kierunek najszybszego spadku.

Zarówno we wzorze (10) jak i w dwóch kolejnych poniżej należy zwrócić uwagę na dobieraną w trakcie uczenia wartość α, która nieodpowiednio dostrojona może powodować znacznie pogorszyć jakość optymalizacji (rys. 41).

Stochastyczny spadek wzdłuż gradientu (ang. *stochastic gradient descent*, SGD) jest ulepszoną wersją spadku wzdłuż gradientu z tą różnicą, że nie używa się wszystkich danych z zasobu, lecz **losowo** wybierana jest pojedyncza obserwacja. Krok jest powtarzany tyle razy ile danych jest w zestawie [39]. Dzięki tej niewielkiej zmianie model zyskuje na szybkości, jest bardziej skalowalny i nie wymaga przydzielenia dużego zasobu pamięci. Algorytm ma postać:

$$\theta_0 = \theta - \alpha \bigtriangledown_\theta J(\theta; x^i; y^i), \tag{11}$$

gdzie:
θ_0 – nowe przybliżenie rozwiązania,
θ – aktualne przybliżenie rozwiązania,
α – współczynnik określający tempo uczenia się algorytmu,
x^i – losowa próbka wyselekcjonowana do obliczeń,
y^i – etykieta przypisana próbce x^i,
$\bigtriangledown_\theta J(\theta)$ – kierunek najszybszego spadku.



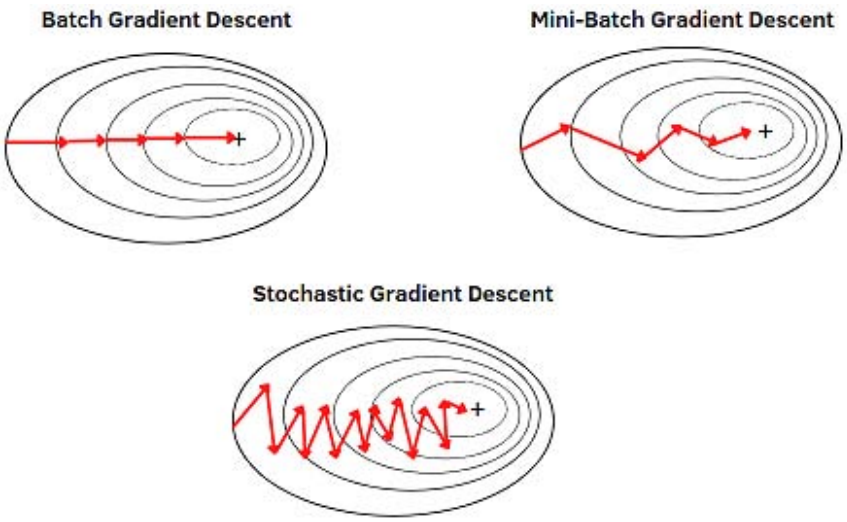
Rysunek 41. Wpływ wartości α na koszt uczenia modelu [38].

Spadek wzdłuż gradientu mini-batch (ang. *Mini-batch gradient descent*) to zmodyfikowany algorytm GD, dla którego dobierana jest mniejsza partia (ang. *mini-batch*) reprezentantów z *n* elementów znajdujących się w zestawie danych. Algorytm wybiera *k* z *n* reprezentantów dostępnych w każdej iteracji, przy czym $k < n$. Dzięki temu wagi są aktualizowane na bieżąco (w przeciwieństwie do GD, gdzie wagi aktualizowane są po przejściu pełnej iteracji na zbiorze) [39]. GD mini-batch jest kompromisem pomiędzy algorytmami GD a SGD. Nie obciąża programu nadmiarowymi obliczeniami (jak GD) i dokonuje dokładniejszej (niż SGD) korekty wag. Algorytm dany jest wzorem:

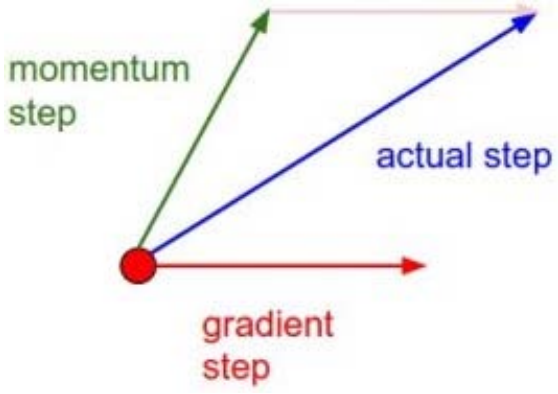
$$\theta_0 = \theta - \alpha \bigtriangledown_\theta J(\theta; x^{i:i+n}; y^{i:i+n}), \tag{12}$$

gdzie:

θ_0 – nowe przybliżenie rozwiązania,
θ – aktualne przybliżenie rozwiązania,
α – współczynnik określający tempo uczenia się algorytmu,
$x^{i:i+n}$ – próbka wyselekcjonowana do obliczeń z aktualizacją dla każdych *n* danych treningowych,
$y^{i:i+n}$ – etykieta przypisana próbce $x^{i:i+n}$,
$\bigtriangledown_\theta J(\theta)$ – kierunek najszybszego spadku.

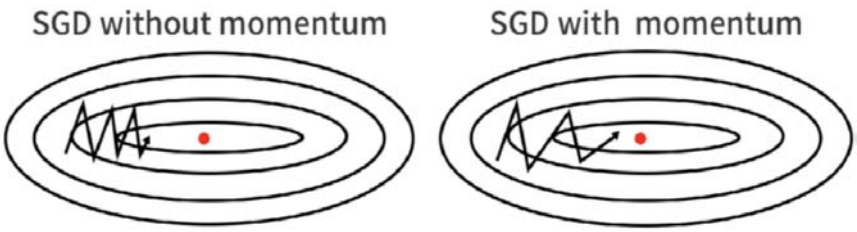


Rysunek 42. Wizualizacja poszukiwania globalnego minimum przez algorytm optymalizacyjny [39].

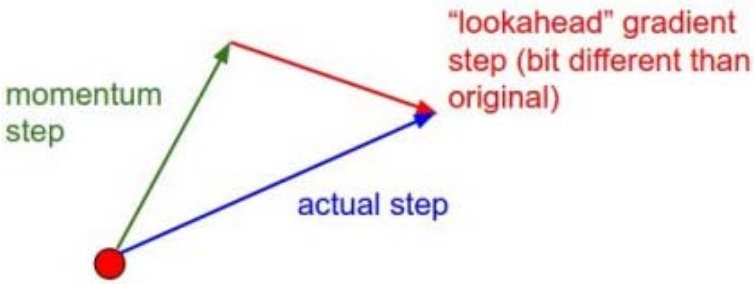


Rysunek 43. Krok szacowany przy użyciu algorytmu Momentum [39].

Rysunek 42 ilustruje wybór kroków w niektórych algorytmach z rodziny GD.



Rysunek 44. Porównanie algorytmów SGD oraz SGD z dodanym Momentum [39].



Rysunek 45. Krok szacowany przez algorytm Nesterov [38].

4.2.2 Momentum

Momentum jest algorytmem opartym na SGD z uwzględnieniem wstecznych wartości gradientów [40]. Wzory pozwalające na wyliczenie Momentum ([2], strona 353) to:

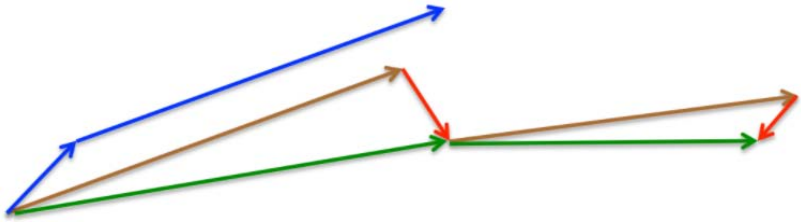
$$\begin{cases} m &= \beta m - \eta \nabla_\theta J(\theta), \\ \theta &= \theta + m, \end{cases} \tag{13}$$

gdzie:

m – wektor momentu,
β – moment (pęd),
η – współczynnik uczenia,
$\nabla_\theta J(\theta)$ – przyspieszenie.

W każdej iteracji od wektora momentu odejmowany jest gradient przemnożony przez współczynnik uczenia (rys. 43). Następuje aktualizacja wag, która decyduje o przyspieszeniu bądź zwolnieniu. Wartością nadzorującą i ewentualnie korygującą nadmierne przyspieszenie jest moment w zakresie od 0 do 1 (wynoszący zazwyczaj 0.9).

Dzięki wprowadzonym zmianom algorytm lepiej radzi sobie z ucieczką z lokalnego minimum (rys. 44), choć nadal może omijać globalne minimum poprzez ciągłe zbliżanie i oddalanie się od wskazanego miejsca.



Rysunek 46. Porównanie kroków w algorytmach Momentum oraz Nesterov [39]. Kolory: niebieski – kroki Momentum, czerwony – korekta kroku w algorytmie Nesterov, zielony – przyspieszenie wynikające z gradientu w algorytmie Nesterov, brązowy – skok w algorytmie Nesterov.

4.2.3 Nesterov

Nesterov jest zmodyfikowanym algorytmem Momentum wzbogaconym o modyfikację polegającą na *pomiarze gradientu w funkcji kosztu nie w lokalnej pozycji θ, ale nieco z przodu w kierunku pędu $\theta + \beta m$* ([2], strona 353). Wzory pozwalające na obliczenie wartości kroków (rys. 45) są następujące ([2], strona 354):

$$\begin{cases} m &= \beta m - \eta \nabla_\theta J(\theta + \beta m), \\ \theta &= \theta + m, \end{cases} \tag{14}$$

gdzie:

m – wektor momentu,
β – moment (pęd),
η – współczynnik uczenia,
$\nabla_\theta J(\theta)$ – przyspieszenie.

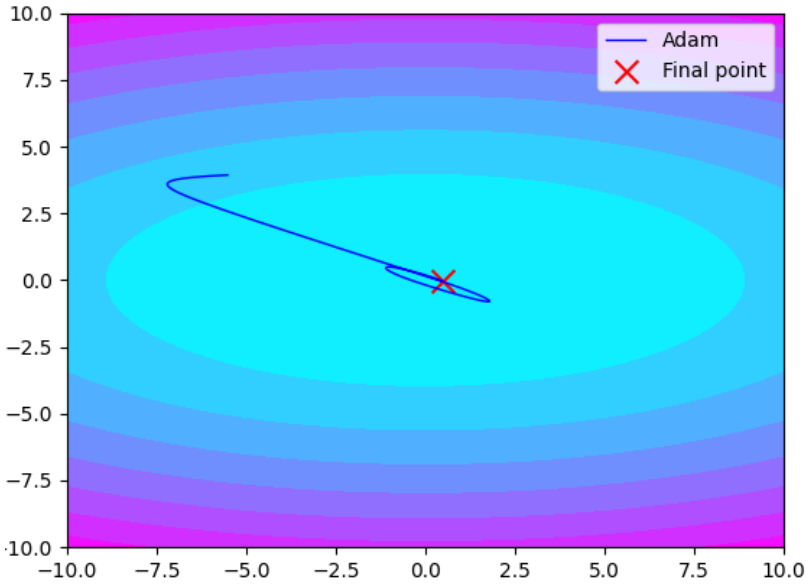
Warto zwrócić uwagę na różnice w zachowaniu algorytmów Momentum oraz Nestrov przy wyliczaniu kolejnych kroków. Na rysunku 46 przedstawiono porównanie kroków w obydwu algorytmach.

4.2.4 RMSProp

RMSProp (ang. *Root Mean Square Propagation*) jest algorytmem, który do aktualizacji wag używa jedynie najnowszych wartości gradientów [36]. Jego zachowanie przypomina działanie algorytmu Momentum z uwzględnieniem redukcji oscylacji wynikających z kolejnych kroków (oscylacje w kształcie litery *W* zostają maksymalnie zredukowane). Algorytm opisany jest równaniami ([2], strona 356):

$$\begin{cases} s &= \beta s + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta), \\ \theta &= \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \varepsilon}, \end{cases} \tag{15}$$

gdzie:

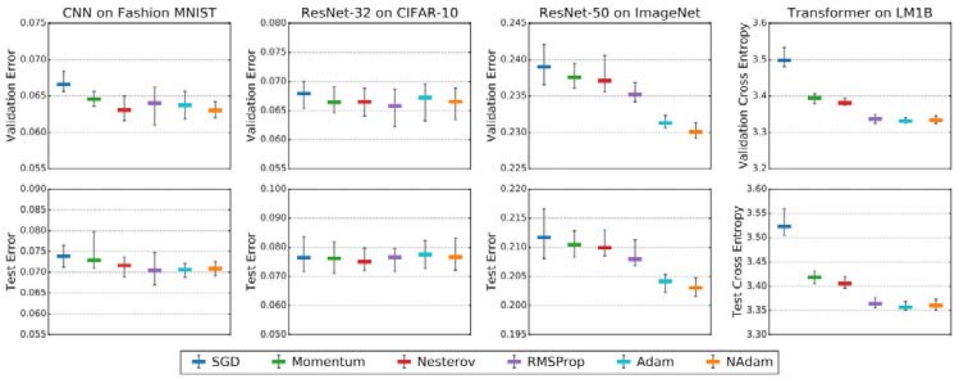


Rysunek 47. Wizualizacja kroków przy użyciu algorytmu Adam [41].

s – wektor,
β – moment (pęd o domyślnej wartości 0,9),
η – współczynnik uczenia,
$\nabla_\theta J(\theta)$ – przyspieszenie.
$\otimes$ – iloczyn Hadamarda – mnożenie odpowiadających sobie elementów,
$\oslash$ – iloraz Hadamarda – dzielenie odpowiadających sobie elementów,
$\sqrt{s+\varepsilon}$ – *człon wygładzający* – uniemożliwia dzielenie przez 0 ([2], strona 355).

4.2.5 Adam

Adam (rys. 47) jest kombinacją algorytmów Momentum oraz RMSProp. Z Momentum *bierze śledzenie rozkładu wykładniczego średniej wcześniejszych gradientów* ([2], strona 357), zaś z RMSProp *śledzenie rozkładu wykładniczego średniej wcześniejszych kwadratów gradientów* ([2], strona 357). Należy pamiętać, że wektory s i m są inicjalizowane zerami, stąd we wzorach uwzględniono odpowiednio skorygowane wartości $\widehat{s}$ oraz $\widehat{m}$. Charakterystyczne cechy obu algorytmów na których bazuje (Momentum oraz RMSProp) są widoczne bezpośrednio w opisujących



Rysunek 48. Porównanie działania sześciu najczęściej używanych optymalizatorów ([42] str. 19).

	Data set	Model	Task	Metric	Batch size	Budget *in epochs*	Approx. run time[3]
P1	Artificial	Noisy quadratic	Minimization	Loss	128	100	< 1 min
P2	MNIST	VAE	Generative	Loss	64	50	10 min
P3	Fashion-MNIST	Simple CNN: *2c2d*	Classification	Accuracy	128	100	20 min
P4	CIFAR-10	Simple CNN: *3c3d*	Classification	Accuracy	128	100	35 min
P5	Fashion-MNIST	VAE	Generative	Loss	64	100	20 min
P6	CIFAR-100	*All-CNN-C*	Classification	Accuracy	256	350	4 h 00 min
P7	SVHN	*Wide ResNet 16-4*	Classification	Accuracy	128	160	3 h 30 min
P8	War and Peace	RNN	Character Prediction	Accuracy	50	200	5 h 30 min

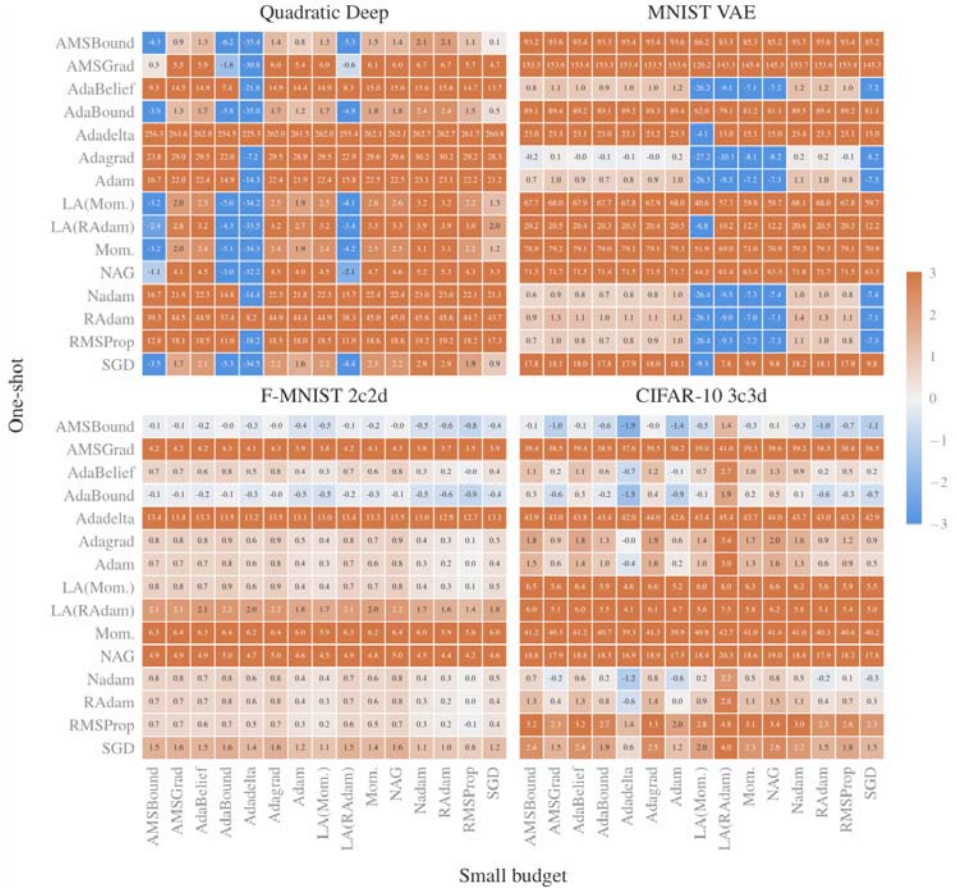
Rysunek 49. Klasyfikacja ośmiu zadań, nad którymi pracowano w ramach porównywania optymalizatorów [42].

go równaniach ([2], strona 357):

$$\begin{cases} m &= \beta_1 m - (1-\beta_1)\nabla_\theta J(\theta), \\ s &= \beta_2 s + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta), \\ \widehat{m} &= \frac{m}{1-\beta_1^t}, \\ \widehat{s} &= \frac{s}{1-\beta_2^t}, \\ \theta &= \theta + \eta \widehat{m} \varnothing \sqrt{\widehat{s} + \varepsilon}. \end{cases} \tag{16}$$

W powyższych równaniach przyjęto następujące oznaczenia:

s, m – wektory,
$\widehat{s}, \widehat{m}$ – wektory skorygowane pod kątem obciążenia (ang. *bias*),
t – numer przebiegu $1, 2, 3, \ldots, n$,
$\otimes$ – iloczyn Hadamarda,
$\varnothing$ – iloraz Hadamarda,
β_1 – parametr pierwszego momentu (o domyślnej wartości $0,9$),



Rysunek 50. Porównanie wydajności piętnastu dostrojonych optymalizatorów dla przykładowych czterech zestawów danych [35].

β_2 – parametr drugiego momentu (o domyślnej wartości $0,9$),
η – współczynnik uczenia (o domyślnej wartości $0,001$),
$\nabla_\theta J(\theta)$ – przyspieszenie,
$\theta = \theta + \eta \widehat{m} \varnothing \sqrt{\widehat{s} + \varepsilon}$ – *człon wygładzający* – uniemożliwia dzielenie przez 0 ([2], strona 355).

4.2.6 Znaczenie wyboru optymalizatora

Wybór właściwego optymalizatora, a potem jego odpowiednie dostrojenie jest sprawą kluczową. O tym jak ważne są to problemy świadczy fakt, że badaniu optymalizatorów poświęca się wiele uwagi. Prowadzone są m.in. badania porównujące optymalizatory. Dla przykładu, w pracy [42]

badano względną wydajność sześciu optymalizatorów. Uzyskane wyniki przedstawia rysunek 48. Z kolei w [35] wskazano osiem problemów, na których sprawdzono działanie optymalizatorów (rys. 49). Na rysunku 50 pokazano porównanie pomiędzy efektywnością niedostrojonych optymalizatorów względem optymalizatorów dostrojonych. Uzyskane wyniki wskazują bezwzględną poprawę efektywności optymalizacji po przełączeniu z dowolnego optymalizatora niedostrojonego na dowolny optymalizator dostrojony.

Na podstawie znajomości wyników prac badających różne optymalizatory można zaznajomić się z ich cechami. Pozwoli to w konsekwencji dobrać odpowiedni optymalizator, właściwy dla rozwiązywanego problemu

5 Podsumowanie

W artykule zostały przedstawione najistotniejsze zagadnienia związane z autoenkoderami. Wyjaśniono pojęcie autoenkodera. Stwierdzono, że jest to sieć neuronowa złożona z mniejszych, współpracujących ze sobą sieci neuronowych zwanych koderami oraz dekoderami. Przedstawiono także zadania, jakie realizują kodery oraz dekodery. W dalszej kolejności omówiono rodzaje enkoderów. Pokrótce zostały opisane autoenkodery wariacyjne, odszumiające, rzadkie, konwolucyjne oraz autoenkodery rekurencyjne. W kontekście autoenkoderów konwolucyjnych omówiono jądro splotowe, filtry będące zestawem jąder splotowych służących do mapowania danych, a także proces redukcji danych, znany pod angielską nazwą *pooling*. W kontekście autoenkoderów konwolucyjnych omówiono jądro splotowe, filtry będące zestawem jąder splotowych służących do mapowania danych, a także proces redukcji danych, znany pod angielską nazwą *pooling*.

W artykule przedstawiono także problemy związane z zastosowaniem autoenkoderów. Najpierw te problemy zdefiniowano, a następnie przedstawiono narzędzia wspomagające ich rozwiązywanie:

- Pierwszym z tych narzędzi jest funkcja strat mierząca błąd działania autoenkodera. Funkcję strat można definiować na wiele sposobów. W artykule zdefiniowano funkcje strat w postaci błędu średniokwadratowego (MSE), pierwiastkowego błędu średniokwadratowego (RMSE), średniego błędu bezwzględnego (MAE), a także w postaci tzw. straty Hubera.
- Drugim narzędziem jest odpowiednio dobrany optymalizator. W artykule zaprezentowano kilka optymalizatorów: GD oraz SGD, Momentum, Nestrov, RMSProp, oraz optymalizator Adam.

Tematyka poruszona w niniejszej pracy pozwala na zapoznanie się z podstawami działania autoenkoderów używanych w uczeniu głębokim. W celu wydajnego szkolenia modeli nie zaleca się bazować na manualnym selekcjonowaniu cech, które będą przekazywane do kolejnych warstw sieci neuronowej, ponieważ liczba cech i ich kombinacji znacznie przewyższa opłacalność poświęconych na to zasobów czasowych. Z pomocą przychodzą odpowiednio dostrojone autoenkodery. Z uwagi na rosnące koszty (czasu, energii, zasobów ludzkich) oraz wymagania (sprzętowe oraz ze strony klientów) optymalizacja procesu staje się niezbędna.

Informacja

Osoby zainteresowane tematyką NLP mogą zaczerpnąć więcej informacji z publikacji „*Natural Language Processing Advancements By Deep Learning: A Survey*" autorstwa Amirsina Torfi i in. [43], dostępnej online pod adresem: `https://arxiv.org/abs/2003.01200/`.

Literatura

[1] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," https://arxiv.org/abs/2003.05991, 2021.

[2] A. Géron, *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow*. Helion, 2020.

[3] D. Foster, *Deep learning i modelowanie generatywne*. Helion, 2021.

[4] E. Kan, "What the heck are vae-gans?" https://towardsdatascience.com/what-the-heck-are-vae-gans-17b86023588a, 2018.

[5] A. Creswell and A. A. Bharath, "Denoising adversarial autoencoders," https://arxiv.org/abs/1703.01220, 2018.

[6] S. Irhum, "Intuitively understanding variational autoencoders," https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf, 2018.

[7] https://thispersondoesnotexist.com/.

[8] https://thiscatdoesnotexist.com.

[9] https://thishorsedoesnotexist.com/.

[10] J. Thompson, "Neural style transfer with swift for tensorflow," https://medium.com/@build_it_for_fun/neural-style-transfer-with-swift-for-tensorflow-b8544105b854, 2019.

[11] H. Liang, L. Yu, G. Xu, B. Raj, and R. Singh, "Controlled autoencoders to generate faces from voices," https://arxiv.org/abs/2107.07988, 2021.

[12] C. Doersch, "Tutorial on variational autoencoders," https://arxiv.org/abs/1606.05908, 2021.

[13] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," https://arxiv.org/abs/1906.02691, 2019.

[14] L. Regenwetter, A. H. Nobari, and F. Ahmed, "Deep generative models in engineering design: A review," https://arxiv.org/abs/2110.10863, 2021.

[15] http://thisxdoesnotexist.com.

[16] T. Park, J.-Y. Zhu, O. Wang, J. Lu, E. Shechtman, A. A. Efros, and R. Zhang, "Swapping autoencoder for deep image manipulation," https://arxiv.org/abs/2007.00653, 2020.

[17] L. Weng, "From autoencoder to beta-vae," https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae, 2018.

[18] K. Cho, "Boltzmann machines and denoising autoencoders for image denoising," https://arxiv.org/abs/1301.3468, 2013.

[19] I. Zenbout, A. Bouramoul, and S. Meshoul, "Stacked sparse autoencoder for unsupervised features learning in pancancer mirna cancer classification," http://ceur-ws.org/Vol-2589/Paper3.pdf, 2020.

[20] A. Ng, "Sparse autoencoder," https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf.

[21] E. Blanco-Mallo, B. Remeseiro, V. Bolón-Canedo, and A. Alonso-Betanzos, "On the effectiveness of convolutional autoencoders on image-based personalized recommender systems," https://arxiv.org/abs/2003.06205, 2020.

[22] R. Kumar, "Faster image classification using tensorflow's graph mode," https://medium.com/artificialis/faster-image-classification-using-tensorflows-graph-mode-67098154808b, 2021.

[23] J. Krohn, G. Beyleveld, and A. Bassens, *Uczenie głębokie i sztuczna inteligencja*. Helion, 2021.

[24] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/.

[25] P. Ganesh, "Types of convolution kernels: Simplified," https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37, 2019.

[26] S. Saha, "A comprehensive guide to convolutional neural networks - the eli5 way," https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018.

[27] "Convolutional autoencoders for image noise reduction," https://towardsdatascience.com/convolutional-autoencoders-for-image-noise-reduction-32fce9fc1763, 2019.

[28] L. Moroney, *Sztuczna inteligencja i uczenie maszynowe dla programistów*. Helion, 2021.

[29] I. Aoukli, "Application of rnn autoencoders: Translation," https://medium.com/@ikhlass.aoukli/application-of-rnn-autoencoders-translation-b4da019e81ea, 2020.

[30] T. Wong and Z. Luo, "Recurrent auto-encoder model for large-scale industrial sensor signal analysis," https://arxiv.org/abs/1807.03710, 2018.

[31] G. Seif, "Understanding the 3 most common loss functions for machine learning regression," https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3, 2019.

[32] "Keras api reference / losses / regression losses," https://keras.io/api/losses/regression_losses/.

[33] A. Opidi, "Pytorch loss functions: The ultimate guide," https://neptune.ai/blog/pytorch-loss-functions, 2021.

[34] P. Sharma, "Pytorch optimizers – complete guide for beginner," https://machinelearningknowledge.ai/pytorch-optimizers-complete-guide-for-beginner/, 2021.

[35] R. Schmidt, F. Schneider, and P. Hennig, "Descending through a crowded valley - benchmarking deep learning optimizers," https://arxiv.org/abs/2007.01547, 2020.

[36] J. Howard and S. Gugger, *Deep learning dla programistów*. Helion, 2021.

[37] "Journey of gradient descent – from local to global," https://laptrinhx.com/journey-of-gradient-descent-from-local-to-global-2829573297/, 2021.

[38] "Neural netrowks 3," https://cs231n.github.io/neural-networks-3/, 2020, stanford University.

[39] G. Tanner, "Understanding optimization algorithms," https://ml-explained.com/blog/gradient-descent-explained, 2021.

[40] S. Weidman, *Uczenie głębokie od zera*. Helion, 2020.

[41] "Adam optimizer," https://machinelearningjourney.com/index.php/2021/01/09/adam-optimizer/.

[42] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," https://arxiv.org/abs/1910.05446, 2020.

[43] A. Torfi, R. A. Shirvani, Y. Keneshloo, N. Tavaf, and E. A. Fox, "Natural language processing advancements by deep learning: A survey," https://arxiv.org/abs/2003.01200, 2020.

Autoencoders. Fundamentals of building efficient machine learning models

Abstract

An autoencoder is a neural network composed of an encoder-decoder pair. The encoder reduces the dimensionality of the data leaving only key features in the model to allow the decoder to reconstruct the input data. Taking into account the internal architecture of autoencoders, a distinction can be made between deterministic autoencoders and probabilistic autoencoders. Only the latter are generative in nature. There are specialised versions of autoencoders corresponding to the subject matter of the machine learning models implemented, for example, de-noising, recursive, convolutional, variational or sparse autoencoders. This paper aims to present the most relevant issues related to autoencoders.

Keywords — machine learning, deep learning, autoencoders, variational autoencoders, denoise autoencoders, sparse autoencoders, convolution autoencoders, recursive autoencoders