



WYDZIAŁ FIZYKI TECHNICZNEJ
I MATEMATYKI STOSOWANEJ

Imię i nazwisko studenta: Kamil Łangowski

Nr albumu: 176553

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Matematyka

Specjalność: Analityk danych

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Metody uczenia głębokiego w analizie danych medycznych

Tytuł pracy w języku angielskim: Deep learning methods in the analysis of medical data

Opiekun pracy: dr hab. Paweł Pilarczyk

OŚWIADCZENIE dotyczące pracy dyplomowej zatytułowanej: Metody uczenia głębokiego w analizie danych medycznych

Imię i nazwisko studenta: Kamil Łangowski

Data i miejsce urodzenia: 16.09.1999, Kościerzyna

Nr albumu: 176553

Wydział: Wydział Fizyki Technicznej i Matematyki Stosowanej

Kierunek: matematyka

Poziom kształcenia: drugi

Forma studiów: stacjonarne

Typ pracy: praca dyplomowa magisterska

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2019 r. poz. 1231, z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t.j. Dz. U. z 2020 r. poz. 85, z późn. zm.),¹ a także odpowiedzialności cywilnoprawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

26.09.2023, Kamil Łangowski

Data i podpis lub uwierzytelnienie w portalu uczelnianym Moja PG

**) Dokument został sporządzony w systemie teleinformatycznym, na podstawie §15 ust. 3b Rozporządzenia MNiSW z dnia 12 maja 2020 r. zmieniającego rozporządzenie w sprawie studiów (Dz.U. z 2020 r. poz. 853). Nie wymaga podpisu ani stempla.*

¹ Ustawa z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce:

Art. 312. ust. 3. W przypadku podejrzenia popełnienia przez studenta czynu, o którym mowa w art. 287 ust. 2 pkt 1–5, rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 312. ust. 4. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 5, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o podejrzeniu popełnienia przestępstwa.

Streszczenie

Celem niniejszej pracy magisterskiej jest opisanie wybranych metod uczenia głębokiego sieci neuronowych w analizie danych medycznych oraz wykonanie eksperymentu polegającego na zastosowaniu wybranych metod uczenia głębokiego do konkretnego zbioru obrazów medycznych. W ramach pracy omawiane są podstawy uczenia maszynowego oraz matematyczne fundamenty, na których opierają się modele uczenia głębokiego. Przedstawiane jest zarówno intuicyjne, jak i formalne podejście do jednokierunkowych sztucznych sieci neuronowych oraz sieci splotowych. W części eksperymentalnej pracy wykorzystywane są metody uczenia głębokiego do klasyfikacji obrazów tkanek jelit, identyfikując tkanki zdrowe oraz chore. Dodatkowo analizowane są obrazy tkanki płucnej, z podziałem na tkanki zdrowe i chore w celu rozróżniania rodzajów nowotworów oraz tkanek zdrowych. Wykonywane są eksperymenty, które pozwalają na zrozumienie potencjału modeli uczenia głębokiego w dziedzinie medycyny. Nieodłączną częścią pracy jest również oprogramowanie, które było utworzone na potrzeby wykonania badań. W pracy są również opisane wykorzystane narzędzia programistyczne.

Słowa kluczowe: uczenie maszynowe, uczenie głębokie, sztuczne sieci neuronowe, sieci splotowe, wykrywanie nowotworów, klasyfikacja obrazów medycznych, Keras, TensorFlow.

Dziedzina nauki i techniki zgodnie z wymogami OECD: 1.1 Matematyka

Abstract

The aim of this master's thesis is to describe selected methods of deep learning in the analysis of medical data and to conduct an experiment involving the application of these selected deep learning methods to a specific set of medical images. The thesis covers the basics of machine learning and the mathematical foundations upon which deep learning models are built. Both intuitive and formal approaches to feedforward artificial neural networks and convolutional networks are presented. In the experimental part of the thesis, deep learning methods are employed to classify images of colon tissues, identifying healthy and diseased tissues. Additionally, images of lung tissue are analyzed, distinguishing between healthy and diseased tissues for the purpose of classifying different types of tumors and healthy tissues. Experiments are conducted to gain an understanding of the potential of deep learning models in the field of medicine. An essential part of the thesis also includes the software developed for the research. The thesis also describes the programming tools used in the study.

Keywords: machine learning, deep learning, artificial neural networks, convolutional networks, tumor detection, medical image classification, Keras, TensorFlow.

The field of science and technology in line with OECD requirements: 1.1 Mathematics

Spis treści

Wstęp	8
1 Preliminaria	10
1.1 Podstawy matematyczne	10
1.2 Uczenie maszynowe	11
1.3 Zadanie klasyfikacji i regresji	12
1.4 Zagadnienie uczenia nadzorowanego	12
1.5 Przetrenowanie i niedotrenowanie modelu	13
1.6 Rodzina rozkładów wykładniczych z dyspersją	13
1.7 Uogólniony model liniowy	16
1.8 Standardowy proces eksploracji danych CRISP-DM	17
2 Wprowadzenie do teorii sztucznych sieci neuronowych i uczenia głębokiego	20
2.1 Jednokierunkowe sztuczne sieci neuronowe	20
2.2 Perceptron	21
2.3 Perceptron wielowarstwowy	22
2.4 Warstwa porzucenia	22
2.5 Jednokierunkowa sieć neuronowa w ujęciu formalnym	23
2.6 Algorytm gradientu prostego ze wstępna propagacją błędu	26
3 Charakterystyka uczenia głębokiego w przetwarzaniu obrazu	30
3.1 Wprowadzenie do zadania rozpoznawania obrazów	30
3.2 Ogólna architektura sieci splotowej	30
3.3 Operator splotu w przetwarzaniu obrazu	31
3.4 Warstwa splotowa	33
3.5 Filtr splotu i mapy cech	35
3.6 Warstwa łącząca	36
3.7 Sieć splotowa w ujęciu formalnym	38
3.8 Przykład architektury sieci splotowej	41
4 Zastosowanie uczenia głębokiego w analizie danych medycznych	43
4.1 Przegląd literatury	43
4.2 Opis zbioru danych	44
4.3 Rozpoznanie potrzeby badawczej	45
4.4 Zrozumienie danych	45
4.5 Przygotowanie danych	46
4.6 Budowa modeli	47
4.7 Ocena modeli pod kątem jakości i efektywności	51
4.8 Eksperymentalne sprawdzenie użyteczności warstwy porzucenia	65
4.9 Eksperiment z inną funkcją aktywacji	66
4.10 Wdrożenie modeli	68
4.11 Dalsze możliwości rozwoju	68
5 Oprogramowanie	70
5.1 Opis bibliotek wykorzystanych w oprogramowaniu	70
5.2 Kod programu	71
Podsumowanie	83
Bibliografia	85

Spis rysunków

1.1 Schemat procesu CRISP-DM	18
2.1 Przykład prostej architektury jednokierunkowej sztucznej sieci neuronowej	21
2.2 Operacja wykonywana w TLU	22
2.3 Przykład wielowarstwowego perceptronu	23
2.4 Warstwa porzucenia	23
2.5 Schemat przykładowej jednokierunkowej sztucznej sieci neuronowej	26
2.6 Potencjalne problemy w algorytmie gradientu prostego	29
3.1 Schemat architektury sieci splotowej	31
3.2 Wyjaśnienie działania splotu	33
3.3 Połączenia rozproszone	34
3.4 Warstwy sieci splotowej z polami receptivejnymi	35
3.5 Współdzielenie parametrów	36
3.6 Dwie mapy cech dla dwóch filtrów splotu	37
3.7 Maksymalizująca warstwa łącząca	37
4.1 Gruczolakorak płuca	44
4.2 Tkanka płucna wolna od nowotworu	44
4.3 Rak płaskonabłonkowy płuca	44
4.4 Tkanka okreżnicy wolna od nowotworu	45
4.5 Gruczolakorak okreżnicy	45
4.6 Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki jelita .	53
4.7 Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki jelita	53
4.8 Macierz pomyłek modelu 1 dla obrazów tkanki jelita	54
4.9 Wykres wartości funkcji straty dla treningu modelu 2 dla obrazów tkanki jelita .	57
4.10 Wykres wartości metryki dokładności dla treningu modelu 2 dla obrazów tkanki jelita	57
4.11 Macierz pomyłek modelu 2 dla obrazów tkanki jelita	58
4.12 Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki płucnej .	60
4.13 Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki płucnej	60
4.14 Macierz pomyłek modelu 1 dla obrazów tkanki płucnej	61
4.15 Wykres wartości funkcji straty dla treningu modelu 2 dla obrazów tkanki płucnej .	63
4.16 Wykres wartości metryki dokładności dla treningu modelu 2 dla obrazów tkanki płucnej	63
4.17 Macierz pomyłek modelu 2 dla obrazów tkanki płucnej	64
4.18 Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki jelita bez warstw porzucenia	66
4.19 Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki jelita bez warstw porzucenia	66
4.20 Wykres wartości funkcji straty dla treningu modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita	67
4.21 Wykres wartości metryki dokładności dla treningu modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita	67
4.22 Macierz pomyłek modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita	68
5.1 Dwie wersje implementacji interfejsu Keras	70

Spis tabel

2.1	Wybrane funkcje aktywacji neuronu stosowane w sieciach	24
4.1	Przebieg treningu modelu 1 dla obrazów tkanki jelita	52
4.2	Przebieg treningu modelu 2 dla obrazów tkanki jelita	56
4.3	Przebieg treningu modelu 1 dla obrazów tkanki płucnej	59
4.4	Przebieg treningu modelu 2 dla obrazów tkanki płucnej	62
4.5	Podsumowanie stworzonych modeli	65

Wstęp

Uczenie głębokie swój początek bierze od najprostszych modeli sieci neuronowych zaproponowanych już w latach 60. XX wieku przez Franka Rosenblatta (zob. [23]). Przez wiele lat modele uczenia głębokiego nie były intensywnie rozwijane. Powód tego stanu rzeczy tkwił głównie w ograniczeniach technologicznych oraz znacznej złożoności obliczeniowej, która była wymagana do trenowania głębokich sieci neuronowych. Sytuacja ta zmieniła się wraz z nadaniem rewolucji cyfrowej i pod koniec XX wieku świat nauki ponownie zwrócił swoją uwagę ku ideom, które wcześniej były często zbyt trudne do wdrożenia. Rozwój sieci neuronowych rozpoczął czwartą rewolucję przemysłową, która trwa do dzisiaj, a jej centralny element stanowi uczenie głębokie, popularniej znane jako sztuczna inteligencja. Dostrzeżono ogromne możliwości uczenia głębokiego w niezliczonych dziedzinach życia codziennego i nauki. Jedną z tych dziedzin jest medycyna.

Medyczne wykorzystanie sztucznych sieci neuronowych ma duży potencjał w diagnozowaniu chorób oraz prognozowaniu ryzyka zachorowania w oparciu o dane pacjenta. Warto podkreślić, że możliwości uczenia głębokiego w diagnostyce nie są ograniczone przez typ oraz wolumen danych. Możliwe jest opieranie modeli na danych tekstowych, takich jak wywiad z pacjentem lub wyniki badań laboratoryjnych. Nie inaczej jest w przypadku danych będącymi obrazami. Mogą być to zarówno obrazy radiologiczne, jak i obrazy histopatologiczne, które można wykorzystać w celu identyfikacji zagrożeń zdrowotnych oraz przekazania odpowiednich powiadomień lekarzom. Z pełną powagą można stwierdzić, że wykorzystanie uczenia głębokiego w medycynie stanowi ogromną nadzieję na poprawę możliwości ratowania ludzkiego życia i zdrowia. Niniejsza praca magisterska stanowi skromny wkład właśnie w tym kierunku.

Celem tej pracy magisterskiej jest opisanie wybranych metod uczenia głębokiego sieci neuronowych w analizie danych medycznych. W szczególności zależy nam na syntetycznym opisie teorii stojącej u podstaw uczenia głębokiego na podstawie różnorodnej literatury. Chcemy przedstawić teorię sztucznych sieci neuronowych w sposób dogłębny. Oznacza to, że w naszym opisie położymy nacisk na stronę intuicyjną omawianych kwestii, a także na rygorystyczny formalizm matematyczny bazujący w części na fundamentalnych dziedzinach matematyki. To holistyczne ujęcie tematu ma na celu zapelnienie pewnej luki w literaturze traktującej o uczeniu głębokim, gdzie z trudem możemy znaleźć pozycje, które w sposób spójny łączyłyby zrozumienie intuicyjne z formalnym aspektem matematycznym. Podczas opracowywania tych fundamentów teoretycznych skorzystaliśmy z różnorodnych źródeł literaturowych, adaptując i integrując fragmenty tekstu z różnych źródeł w celu stworzenia spójnego i klarownego omówienia teorii sztucznych sieci neuronowych, które łączy aspekt intuicyjny z aspektem formalnym. Nasze rozważania koncentrujemy na możliwościach uczenia głębokiego w przetwarzaniu obrazu, a więc dążymy do pełnego opisu sieci splotowych, wychodząc od uogólnionego modelu liniowego, a następnie jednokierunkowych sztucznych sieci neuronowych. Starania o pełne zrozumienie podstaw teoretycznych są umotywowane kolejnym celem pracy, którym jest praktyczne zastosowanie omawianych modeli.

Praktyczna realizacja modeli poprzedzona omówieniem teoretycznym daje pewien wgląd w możliwości uczenia głębokiego. W naszych eksperymentalnych rozważaniach zadecydowaliśmy o skupieniu się na danych medycznych w formie obrazów. W tym celu wyszukaliśmy ogólnodostępny, duży zbiór danych graficznych, który zawiera w sobie obrazy histopatologiczne tkanek płucnej i jelit. Przedstawiamy różne podejścia do budowy modeli sieci splotowych i analizy danych, dając przy tym do osiągnięcia jak najwyższej wydajności, analizując wpływ różnych czynników na rezultaty tych modeli oraz wskazując na potencjalne możliwości rozwoju naszych badań. Nie pomijamy przy tym kwestii technicznych (programistycznych), które także opisujemy.

Strukturę pracy tworzy pięć rozdziałów. Rozdział 1 poświęcony jest pojęciom wstępny i przydatnym w opisie aspektów teoretycznych znajdujących się w pracy, a także samej dziedziny uczenia maszynowego. W rozdziale 2 opisujemy podstawowy model uczenia głębokiego, jakim jest jednokierunkowa sztuczna sieć neuronowa, która jest modelem wyjściowym na drodze do bardziej zaawansowanych modeli. Rozdział 3 traktuje o sieciach splotowych, a zatem o zastosowaniu uczenia głębokiego w przetwarzaniu obrazu. Można powiedzieć, że rozdział 3 stanowi najważniejszą część pracy w obszarze teoretycznym. Rozdział 4 rozpoczyna część eksperymentalną pracy. W tym rozdziale wykonujemy opis wszystkich etapów wykonanego projektu badawczego w dziedzinie eksploracyjnej analizy danych z wykorzystaniem metod uczenia głębokiego. W pracy nad problemem badawczym podążamy za schematem CRISP-DM (opisanym w rozdziale 1), którego kolejne kroki stanowią kolejne podrozdziały rozdziału 4. W rozdziale 5 wykonujemy opis narzędzi programistycznych wykorzystanych w pracy, a także umieszczamy kod programu, który wykonaliśmy na potrzeby pracy.

Rozdział 1

Preliminaria

W niniejszym rozdziale opisane zostały pojęcia i zagadnienia pojawiające się w rozdziale 2 i w rozdziale 3, których znajomość jest istotna dla zrozumienia wprowadzonej teorii oraz zastosowań.

1.1 Podstawy matematyczne

W tym podrozdziale przedstawiamy kilka definicji dotyczących podstaw matematyki, które są istotne dla zrozumienia pewnych treści z tego i kolejnych rozdziałów. Niniejszy podrozdział powstał na podstawie [5, 6, 7, 11, 21].

Definicja 1.1 (Przestrzeń probabilistyczna) Niech Ω będzie przestrzenią zdarzeń elementarnych ozn. $A \subset \Omega$. Niech \mathcal{F} będzie σ -ciałem podzbiorów zbioru Ω . Niech P będzie dowolną funkcją określona na σ -ciele zdarzeń $\mathcal{F} \subset 2^\Omega$, spełniającą warunki

1. $P: \mathcal{F} \rightarrow \mathbb{R}_+$;
2. $P(\Omega) = 1$;
3. Jeśli $A_i \in \mathcal{F}$, $i = 1, 2, \dots$, oraz $A_i \cap A_j = \emptyset$ dla $i \neq j$, to $P(\bigcup_{i=1}^n A_i) = \sum_{i=1}^{\infty} P(A_i)$.

P nazywamy prawdopodobieństwem. Wówczas trójkę (Ω, \mathcal{F}, P) nazywamy *przestrzenią probabilistyczną*.

Definicja 1.2 (Zmienna losowa) Niech (Ω, \mathcal{F}, P) będzie przestrzenią probabilistyczną. Mówimy, że funkcja rzeczywista

$$X: \Omega \rightarrow \mathbb{R} \tag{1.1}$$

jest *zmienną losową*, jeżeli dla dowolnego zbioru borelowskiego $B \in \mathcal{B}_{\mathbb{R}}$ jego przeciwbraz $X^{-1}(B) = \{\omega \in \Omega: X(\omega) \in B\}$ jest elementem σ -ciała \mathcal{F} .

Definicja 1.3 (Wektor losowy) Niech (Ω, \mathcal{F}, P) będzie przestrzenią probabilistyczną. Mówimy, że funkcja rzeczywista

$$\mathbf{X}: \Omega \rightarrow \mathbb{R}^n \tag{1.2}$$

jest *n-wymiarowym wektorem losowym*, jeżeli dla dowolnego zbioru borelowskiego $B \in \mathcal{B}_{\mathbb{R}^n}$ jego przeciwbraz $\mathbf{X}^{-1}(B) = \{\omega \in \Omega: \mathbf{X}(\omega) \in B\} \in \mathcal{F}$.

Definicja 1.4 (Rozkład prawdopodobieństwa) Niech $X: (\Omega, \mathcal{F}, P) \rightarrow \mathbb{R}$ będzie zmienną losową. *Rozkładem prawdopodobieństwa* zmiennej losowej X nazywamy miarę probabilistyczną μ na $(\mathbb{R}, \mathcal{B}_{\mathbb{R}})$ zdefiniowaną:

$$\mu(B) = P(X^{-1}(B)) = P(\{\omega \in \Omega: X(\omega) \in B\}). \tag{1.3}$$

Definicja 1.5 (Wartość oczekiwana) Niech X będzie zmienną losową określona na przestrzeni probabilistycznej (Ω, \mathcal{F}, P) . *Wartością oczekiwana* zmiennej losowej X nazywamy

$$\int_{\Omega} X(\omega) dP(\omega), \tag{1.4}$$

o ile ta całka istnieje, którą rozumiemy jako całkę Lebesgue'a. Całkę tę oznaczamy w skrócie $\mathbb{E}[X]$.

Definicja 1.6 (Wariancja) Niech X będzie całkowalną z kwadratem zmienną losową określona na przestrzeni probabilistycznej (Ω, \mathcal{F}, P) . Wariancją zmiennej losowej X nazywamy liczbę

$$\text{Var}(X) = \mathbb{E}[X - \mathbb{E}[X]]^2. \quad (1.5)$$

Definicja 1.7 (Softmax) Niech \mathbf{x} będzie cechami modelu (model – zob. podrozdz. 1.4, cechy – zob. podrozdz. 1.4 i 1.7), $\boldsymbol{\theta}^{(k)}$ wektorem parametrów modelu (zob. 1.4) dla danej klasy, a $1 \leq k \leq K \in \mathbb{N}$ prognozowaną klasą. Wówczas funkcja $f_k: \mathbb{R} \rightarrow [0; 1]$ określona wzorem

$$f_k(\mathbf{x}^\top \boldsymbol{\theta}^{(k)}) = \frac{\exp(\mathbf{x}^\top \boldsymbol{\theta}^{(k)})}{\sum_{j=1}^K \exp(\mathbf{x}^\top \boldsymbol{\theta}^{(j)})} \quad (1.6)$$

nazywana jest funkcją *softmax* i określa ona prawdopodobieństwo przynależności obserwacji do klasy k . Prognoza klasyfikatora *softmax* odbywa się przez operację

$$\hat{y} = \arg \max_k \{f_k(\mathbf{x}^\top \boldsymbol{\theta}^{(k)})\}. \quad (1.7)$$

Definicja 1.8 (Binarna entropia krzyżowa) Niech $y_\theta^{(i)} \in \{0, 1\}$ będzie rzeczywistą etykietą i -tej obserwacji, przy czym i $y_\theta = [y_\theta^{(1)}, \dots, y_\theta^{(i)}]$. Niech $\hat{y}_\theta^{(i)} \in [0; 1]$ będzie prawdopodobieństwem przynależności i -tej obserwacji do jednej z dwóch klas i $1 \leq i \leq N$. Niech $N \in \mathbb{N}$ będzie liczbą obserwacji, $\theta \in \Theta \subset \mathbb{R}^K$ pewną przestrzenią parametrów modelu, dla $K \in \mathbb{N}$. *Binarna entropia krzyżowa* (rozumiana jako funkcja kosztu) to funkcja $H_\theta: \{0, 1\} \times [0; 1] \rightarrow \mathbb{R}$, wyrażająca się wzorem

$$H_\theta(y_\theta, \hat{y}_\theta) = -\frac{1}{N} \sum_{i=1}^N \left(y_\theta^{(i)} \ln \left(\hat{y}_\theta^{(i)} \right) + (1 - y_\theta^{(i)}) \ln \left(1 - \hat{y}_\theta^{(i)} \right) \right). \quad (1.8)$$

Definicja 1.9 (Kategorialna entropia krzyżowa) Niech $y_\theta^{(i)} \in \{0, 1\}$ będzie rzeczywistą etykietą obserwacji i i $y_\theta = [y_\theta^{(1)}, \dots, y_\theta^{(i)}]$. Niech $p_{k,\theta}^{(i)} \in [0; 1]$ będzie prawdopodobieństwem przynależności obserwacji i do klasy $1 \leq k \leq K$, a także $p_\theta^{(i)} = [p_{1,\theta}^{(i)}, \dots, p_{K,\theta}^{(i)}]$ dla $1 \leq i \leq N$ i $K \in \mathbb{N}$. Niech $N \in \mathbb{N}$ będzie liczbą obserwacji, $\theta \in \Theta \subset \mathbb{R}^K$ pewną przestrzenią parametrów modelu. *Kategorialna entropia krzyżowa* (rozumiana jako funkcja kosztu) to funkcja $H_\theta: \{0, 1\}^N \times [0; 1]^k \rightarrow \mathbb{R}$ wyrażająca się wzorem

$$H_\theta(y_\theta^{(i)}, p_\theta^{(i)}) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^C \left(y_\theta^{(i)} \ln \left(p_{k,\theta}^{(i)} \right) \right). \quad (1.9)$$

Zwróćmy uwagę, że dla $K = 2$ otrzymujemy wzór binarnej entropii krzyżowej.

1.2 Uczenie maszynowe

W niniejszym podrozdziale opisujemy, czym jest dziedzina uczenia maszynowego, na jakie pytania poszukuje odpowiedzi, a także czym w ogólności jest proces uczenia się maszyny. Podrozdział powstał na podstawie [9, 19].

Gdy mówimy o uczeniu maszynowym, ważne jest odróżnienie, czy odnosi się to do całej dziedziny sztucznej inteligencji, czy jedynie do samego procesu uczenia się maszyny (rozumianego ogólnie jako program komputerowy). Dziedzina uczenia maszynowego jest bardzo obszerna i interdyscyplinarna, łączy ze sobą matematykę, statystykę i informatykę.

Dziedzinę nauki można scharakteryzować, formułując główne pytania, na jakie stara się znaleźć odpowiedź. W przypadku uczenia maszynowego dążymy do znalezienia odpowiedzi na pytanie:

„Jak stworzyć system komputerowy, który będzie automatycznie ulepszał się (uczył) wraz z doświadczeniem, jakie zdobywa, oraz czy istnieją podstawowe prawa, które rządzą całym procesem uczenia się tego systemu?”

To pytanie obejmuje wiele różnych zadań, które możemy rozważyć w kontekście uczenia maszynowego. Na przykład, jak można skorzystać z informacji dotyczących pewnych chorób, aby w przyszłości wykonywać bardziej skuteczne diagnozy u pacjentów lub dostosowywać bardziej odpowiednio metody leczenia?

Sam proces uczenia się maszyny można określić następująco:

„Program komputerowy uczy się z doświadczenia E w odniesieniu do pewnej klasy zadań T i miary wydajności P , jeśli jego skuteczność w zadaniach T , mierzona za pomocą miary wydajności P , poprawia się wraz z doświadczeniem E .”

Powyższą definicję zilustrujemy poprzez przykład. Założymy, że naszym celem jest stworzenie klasyfikatora, który będzie rozróżniał nowotwory złośliwe od łagodnych. Wówczas, w kontekście przytoczonej definicji:

- Doświadczeniem E będzie proces klasyfikacji kolejnych nowotworów na złośliwe i łagodne.
- Zadaniem T będzie zaklasyfikowanie danego nowotworu jako złośliwy lub łagodny.
- Miara wydajności P będzie liczbą nowotworów zaklasyfikowanych poprawnie, czyli prawidłowo zidentyfikowanych jako złośliwe lub łagodne. Otrzymujemy tu informację zwrotną o poprawności wykonanej identyfikacji.

W ten sposób umożliwiona jest ciągła poprawa programu. System nie tylko wykonuje klasyfikacje (rozpoznanie rodzaju nowotworu), ale również otrzymuje informacje zwrotne dotyczące ich poprawności. Ten mechanizm informacji zwrotnej napędza iteracyjne doskonalenie zdolności klasyfikacyjnych programu. Na przykład program może przechowywać i analizować wcześniej zdobyte dane, porównując je z nowymi przypadkami, aby poprawić swoją dokładność w miarę upływu czasu poprzez odpowiednie dostrojenie parametrów wykorzystywanego modelu klasyfikacji. Ten proces iteracyjnego uczenia zapewnia, że program staje się coraz bardziej zdolny do diagnozowania pacjentów. Wprowadzenie takich mechanizmów iteracyjnego uczenia oznacza znaczący postęp w precyzyji diagnozowania oraz opiece nad pacjentami.

1.3 Zadanie klasyfikacji i regresji

Niniejszy podrozdział zawiera krótki opis dwóch najbardziej podstawowych zadań nadzorowanego uczenia maszynowego, jakimi są klasyfikacja i regresja. Powstał on w oparciu o [9].

Zadania związane z uczeniem maszynowym (oznaczone w podrozdziale 1.2 przez T) zazwyczaj są opisywane w kontekście tego, w jaki sposób system uczący się powinien przetwarzać przykład. Przykładem jest zbiór cech, które zostały zmierzone ilościowo lub jakościowo z pewnego obiektu lub zdarzenia, które chcemy, aby system uczący się przetwarzał. Przykład możemy reprezentować jako wektor $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^q$, $q \in \mathbb{N}$, gdzie każdy element x_i wektora \mathbf{x} to inna cecha. Na przykład cechy obrazu to zazwyczaj wartości pikseli na obrazie.

Wiele rodzajów zadań może być rozwiązywanych z wykorzystaniem uczenia maszynowego. Poniżej przedstawiamy dwa najpopularniejsze.

- Klasifikacja (ang. *classification*): W tego typu zadaniu program komputerowy ma za zadanie określić, do której z k kategorii należy pewne wejście. Aby rozwiązać to zadanie, zazwyczaj wymaga się od algorytmu uczenia się stworzenia funkcji $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$. Gdy $Y = f(\mathbf{x})$, model przyporządkowuje wejście opisane wektorem \mathbf{x} do kategorii zidentyfikowanej za pomocą numerycznego kodu Y .
- Regresja (ang. *regression*): W tego typu zadaniu program komputerowy ma za zadanie przewidzieć wartość numeryczną na podstawie pewnego wejścia. Aby rozwiązać to zadanie, algorytm uczenia jest proszony o wygenerowanie funkcji $f: \mathbb{R}^n \rightarrow \mathbb{R}$. Ten rodzaj zadania jest podobny do klasyfikacji, z tą różnicą, że format wyniku jest inny.

1.4 Zagadnienie uczenia nadzorowanego

W tym podrozdziale wyjaśniamy czym jest zagadnienie uczenia nadzorowanego oraz jaki jest jego cel. Podrozdział powstał na podstawie [5, podrozdz. 1.2].

Sztuczne sieci neuronowe, stanowiące temat tej pracy, należą do modeli uczenia maszynowego w dziedzinie uczenia nadzorowanego (ang. *supervised learning*). Zatem opis sztucznych sieci neuronowych poprzedzimy opisem zadania uczenia nadzorowanego. W kontekście sztucznych sieci neuronowych rozważać możemy zarówno zadania klasyfikacji, jak i zadania regresji.

W ramach procesu uczenia nadzorowanego dąży się do znalezienia pewnego „optimalnego odwzorowania” między zmiennymi objaśniającymi a zmiennymi objaśnianymi, korzystając z dostępnego zbioru treningowego postaci $\mathcal{L} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Zbiór treningowy zawiera $N \in \mathbb{N}$ realizacji zmiennych objaśnianych (y_i) i objaśniających (\mathbf{x}_i), realizacje te nazywane są także obserwacjami. Wektor \mathbf{x}_i określany jest jako wektor cech (ang. *feature vector*), a wartość y_i jako etykieta tego wektora (ang. *label*).

Celem uczenia nadzorowanego jest wykorzystanie zbioru treningowego w do stworzenia modelu, a więc przekształcenia, który na podstawie wektora cech \mathbf{x}_i jako wejścia generuje informacje

pozwalać wywnioskować etykietę y_i dla tego wektora cech.

Warto nadmienić, że z punktu widzenia probabilistycznego jesteśmy zainteresowani poznaniem rozkładu warunkowego $P(y_i|\mathbf{x}_i, \boldsymbol{\theta})$, gdzie $\boldsymbol{\theta}$ jest wektorem parametrów modelu.

1.5 Przetrenowanie i niedotrenowanie modelu

Niniejszym podrozdziale omawiamy dwóch głównych problemów związanych z treningiem modeli uczenia maszynowego – przetrenowania i niedotrenowania modelu. Przedstawiamy krótkie techniki pozwalające ograniczyć ryzyko wystąpienia tych zjawisk. Podrozdział powstał w oparciu o [7, rozdz. 1].

Przetrenowanie modelu lub nadmierne dopasowanie (ang. *overfitting*) oznacza, że model sprawdza się dobrze jedynie w przypadku danych uczących, ale uogólnienie modelu na przypadki wykraczające poza zbiór testowy nie sprawuje się zbyt dobrze.

Skomplikowane modele, jak sztuczne sieci neuronowe są w stanie wykrywać subtelne wzorce w danych, ale jeżeli zbiór danych jest zaszumiony (duża losowość danych) lub zbyt mały, to taki model będzie wykrywał wzorce nie w użytecznych danych, lecz w szумie. Wówczas wyuczonego modelu nie da się generalizować na nowe próbki.

Istnieją techniki ograniczenia modelu, w celu jego uproszczenia, a tym samym zmniejszenia ryzyka przetrenowania. Taki proces ograniczania modelu nosi nazwę regularyzacji (ang. *regularization*). Przykładowe sposoby przeciwdziałania przetrenowaniu danych uczących to:

- Uproszczenie modelu przez wybór modelu zawierającego mniej parametrów (np. zamiast modelu wielomianowego wykorzystanie modelu liniowego).
- Zmniejszenie liczby atrybutów w danych uczących.
- Zwiększenie liczby przykładów w danych uczących.
- Zmniejszenie zaszumienia danych uczących (np. usunięcie błędnych danych i elementów odstających).
- Regularyzacja hiperparametrów modelu (zob. poniżej).

Hiperparametry modelu to parametry algorytmu uczącego. Nie są one modyfikowane przez sam algorytm uczący, ale są wyznaczane przed rozpoczęciem procesu uczenia i w jego trakcie ich wartości pozostają stałe. Regularyzacja hiperparametrów modelu polega na dopasowywaniu wartości parametrów algorytmu uczącego. Proces regularyzacji hiperparametrów obejmuje testowanie różnych wartości hiperparametrów oraz śledzenie efektywności modelu na zbiorach treningowych, walidacyjnych i testowych. Głównym celem jest wyselekcjonowanie hiperparametrów, które osiągają równowagę między nauką na danych treningowych a zdolnością do skutecznego uogólniania na nowe dane.

Niedotrenowanie modelu (ang. *underfitting*) występuje, gdy model jest zbyt prosty, aby wyuczyć się struktur danych uczących. Oznacza to, że wybrany model za bardzo upraszcza rzeczywistość. Na przykład w przypadku wyboru modelu liniowego, który zakłada liniowość między zmiennymi objaśniającymi a zmienną objasnianą, gdy w rzeczywistości zależność ta jest nielinowa. Przykładowe sposoby na przeciwdziałanie niedotrenowaniu danych uczących to:

- Wybór bardziej skomplikowanego modelu np. z większą liczbą parametrów.
- Zwiększenie liczby cech danych uczących branych pod uwagę podczas budowy modelu.
- Zmniejszenie ograniczenia modelu (np. zredukowanie hiperparametrów regularyzacji).

1.6 Rodzina rozkładów wykładniczych z dyspersją

W niniejszym podrozdziale omawiamy fundamenty matematyczne, na jakich opiera się uogólniony model liniowy, a w konsekwencji sztuczna sieć neuronowa. Przedstawiamy definicję rodziny rozkładów wykładniczych z dyspersją wraz z przykładem rozkładu prawdopodobieństwa wchodzącego w skład tej rodziny. Wprowadzimy też notację dla definiowanych obiektów, którą wykorzystywać będziemy w dalszej części pracy. Podrozdział powstał w oparciu o [31, rozdz. 2].

Zacznijmy od definicji dziedziny efektywnej.

Definicja 1.10 (Dziedzina efektywna) Niech ν będzie σ -skońzoną miarą na \mathbb{R} , niech $a: \mathbb{R} \rightarrow \mathbb{R}$ i $T: \mathbb{R} \rightarrow \mathbb{R}^k$ będą mierzalnymi funkcjami dla $k \in \mathbb{N}$. Rozważmy wektor parametrów

$\boldsymbol{\theta} \in \mathbb{R}^k$ nazywany także parametrem kanonicznym. Wówczas *dziedzina efektywna* (ang. *effective domain*) to zbiór

$$\Theta = \left\{ \boldsymbol{\theta} \in \mathbb{R}^k; \int_{\mathbb{R}} \exp(\boldsymbol{\theta}^\top T(x) + a(x)) d\nu(x) < \infty \right\} \subseteq \mathbb{R}^k. \quad (1.10)$$

W dalszej części pracy zakładając będziemy, że dziedzina efektywna jest wypukła oraz posiada niepuste wnętrze, które oznaczamy jako $\mathring{\Theta}$.

Definicja 1.11 (Funkcja generująca kumulanty) Niech $\boldsymbol{\theta} \in \Theta$, wówczas przy założeniach z definicji 1.10 *funkcja generująca kumulanty* (ang. *cumulant function*) wyraża się wzorem

$$\kappa(\boldsymbol{\theta}) = \ln \left(\int_{\mathbb{R}} \exp(\boldsymbol{\theta}^\top T(x) + a(x)) d\nu(x) \right) \quad (1.11)$$

Przedstawimy teraz ważne własności funkcji generującej kumulanty. Niech Y będzie zmienną losową, pierwsze dwa momenty zmiennej losowej Y , a więc wartość oczekiwana i wariancja, dane są kolejno przez:

$$\mu = \mathbb{E}_{\boldsymbol{\theta}}[Y] = \nabla_{\boldsymbol{\theta}} \kappa(\boldsymbol{\theta}) = \kappa'(\boldsymbol{\theta}) \quad (1.12)$$

i

$$\text{Var}_{\boldsymbol{\theta}}(Y) = \nabla_{\boldsymbol{\theta}}^2 \kappa(\boldsymbol{\theta}) = \kappa''(\boldsymbol{\theta}) > 0, \quad (1.13)$$

gdzie $\nabla_{\boldsymbol{\theta}}$ to gradient, a $\nabla_{\boldsymbol{\theta}}^2$ to hesjan względem wektora $\boldsymbol{\theta} \in \Theta$.

Z funkcją generującą kumulanty bezpośrednio związana jest kanoniczna funkcja związku.

Definicja 1.12 (Kanoniczna funkcja związku) Niech κ będzie funkcją generującą kumulanty. Wtedy *kanoniczna funkcja związku* (ang. *canonical link*) wyraża się wzorem

$$h = (\kappa')^{-1}. \quad (1.14)$$

Inaczej możemy wyrazić tę zależność za pomocą

$$h(\mu) = h(\mathbb{E}_{\boldsymbol{\theta}}[Y]) = \boldsymbol{\theta}, \quad (1.15)$$

dla $\boldsymbol{\theta} \in \Theta$.

Definiujemy funkcję wariancji, która zawiera w sobie funkcję generującą kumulanty oraz kanoniczną funkcję związku.

Definicja 1.13 (Funkcja wariancji) Niech κ będzie funkcją generującą kumulanty, h kanoniczną funkcją związku, a μ pierwszym momentem zmiennej losowej Y . Wtedy *funkcja wariancji* (ang. *variance function*) zmiennej losowej Y zdefiniowana jest jako

$$V(\mu) = (\kappa'' \circ h)(\mu) = \text{Var}_{\mu}(Y). \quad (1.16)$$

Definicja 1.14 (Dualna przestrzeń parametryczna) Założymy, że funkcja generująca kumulanty κ jest wypukła na wnętrzu dziedziny efektywnej $\mathring{\Theta}$. Wówczas *dualna przestrzeń parametryczna* (ang. *dual parameter space*) to zbiór

$$\mathcal{M} = \nabla_{\boldsymbol{\theta}} \kappa(\mathring{\Theta}) = \left\{ \nabla_{\boldsymbol{\theta}} \kappa(\mathring{\Theta}); \boldsymbol{\theta} \in \mathring{\Theta} \right\} \subseteq \mathbb{R}^k. \quad (1.17)$$

Założenie o wypukłości funkcji generującej kumulanty implikuje istnienie odwzorowania jednoznacznego pomiędzy Θ i \mathcal{M} (dowód tego faktu można znaleźć w [31]). Ponadto mamy $\mu = \mathbb{E}_{\boldsymbol{\theta}}[Y] \in \mathcal{M}$.

W celu prostszych rozważań, w dalszej części ograniczamy zdefiniowane obiekty do skalarnego parametru kanonicznego $\boldsymbol{\theta}$, a więc do $k = 1$. Parametr taki zapisywać będziemy jako θ . Ponadto, za mierzalną funkcję T z definicji 1.10 i 1.11 przyjmujemy funkcję indentycznościową. Oczywiście poniższe rozważania można uogólnić na przypadek wielowymiarowy, przyjmując wektorowy parametr kanoniczny.

Wybieramy rodzinę σ -skończonych miar ν_{ω} na \mathbb{R} oraz mierzalne funkcje $a_{\omega}: \mathbb{R} \rightarrow \mathbb{R}$ dla danego zbioru indeksów $\mathcal{W} \ni \omega$, który jest podzbiorem \mathbb{R}_+ . Założymy, że mamy ω -niezależną skalowaną funkcję generującą kumulanty κ na zbiorze indeksów \mathcal{W} , tzn.

$$\kappa(\theta) = \frac{1}{\omega} \left(\ln \int_{\mathbb{R}} \exp(\theta x + a_{\omega}(x)) d\nu_{\omega}(x) \right), \quad (1.18)$$

dla $\theta \in \Theta$ i $\omega \in \mathcal{W}$, gdzie Θ to dziedzina efektywna dla $\omega = 1$. To pozwala nam rozważyć funkcje rozkładów prawdopodobieństwa

$$\begin{aligned} dF(x; \theta, \omega) &= f(x; \theta, \omega) d\nu_\omega(x) \\ &= \exp(\theta x - \omega\kappa(\theta) + a_\omega(x)) d\nu_\omega(x) \\ &= \exp(\omega(\theta y - \kappa(\theta)) + a_\omega(\omega y)) d\nu_\omega(\omega y), \end{aligned} \quad (1.19)$$

gdzie $y = x/\omega$. Poprzez nieco odmienny sposób przeparametryzowania funkcji $a_\omega(\omega \cdot)$ oraz σ -skończonych miar $\nu_\omega(\omega \cdot)$, w zależności od konkretnej struktury wybranych σ -skończonych miar, dochodzimy do następującej rodziny rozkładów wykładniczych z dyspersją (ang. *exponential dispersion family*, EDF).

Definicja 1.15 (Rodzina rozkładów wykładniczych z dyspersją) Niech Y będzie zmieniąną losową, $\kappa: \Theta \rightarrow \mathbb{R}$ funkcją generującą kumulanty, $\theta \in \Theta$ parametrem kanonicznym w dziedzinie efektywnej, $v > 0$ danym parametrem wagi, $\varphi > 0$ parametrem zwanym parametrem dyspersji, $a(\cdot; \cdot)$ funkcją normalizacji niezależną od parametru θ . Wtedy *Rodzina rozkładów wykładniczych z dyspersją* jest dana przez gęstości postaci

$$Y \sim f(y; \theta, v/\varphi) = \exp\left(\frac{y\theta - \kappa(\theta)}{\varphi/v} + a(y; v/\varphi)\right). \quad (1.20)$$

Parametry wagi $v > 0$ i dyspersji $\varphi > 0$ zawsze występują jako stosunek $\omega = v/\varphi \in \mathcal{W}$. Zastosowanie funkcji normalizacji a powoduje, że f całkuje się do 1.

Dyspersja w kontekście Rodziny rozkładów wykładniczych z dyspersją odnosi się do różnorodności i zmienności danych. Niektóre dane są bardziej zróżnicowane, a inne mniej. Na przykład dane o rozproszeniu zbliżonym do zera są bardziej skoncentrowane wokół średniej, podczas gdy dane o większym rozproszeniu mają większy rozrzut od średniej. Rodzina rozkładów wykładniczych z dyspersją pozwala na dostosowanie modeli do różnych poziomów dyspersji w danych poprzez wybór odpowiedniego elementu tej rodziny, a więc rozkładu prawdopodobieństwa.

Wprowadzone dotąd obiekty mogą sprawiać wrażenie abstrakcyjnych, jednakże w zastosowaniu sprowadzają się do względnie prostych funkcji dla popularnych rozkładów prawdopodobieństwa.

Przedstawimy teraz przykład jednego z rozkładów z wykładniczej rodziny rozkładów dyspersyjnych – rozkładu dwumianowego.

Dla rozkładu dwumianowego z parametrami $p \in (0; 1)$ i $n \in \mathbb{N}$ wybieramy miarę liczącą na $\{0, 1/n, \dots, 1\}$ z $\omega = n$. Wówczas wykonujemy następujące wybory:

- funkcja normalizacji: $a(y) = \ln(n_y)$;
- funkcja generująca kumulanty: $\kappa(\theta) = \ln(1 + e^\theta)$ oraz $p = \kappa'(\theta) = \frac{e^\theta}{1 + e^\theta}$;
- kanoniczna funkcja związku $\theta = h(p) = \ln\left(\frac{p}{1-p}\right)$;

dla dziedziny efektywnej $\Theta = \mathbb{R}$ i dualnej przestrzeni parametrów $\mathcal{M} = (0; 1)$.

Po przyjęciu takich wyborów mamy gęstość w postaci

$$f(y; \theta, n) = \binom{n}{ny} \exp(n(\theta y - \ln(1 + e^\theta))) = \binom{n}{ny} \left(\frac{e^\theta}{1 + e^\theta}\right)^{ny} \left(\frac{1}{1 + e^\theta}\right)^{n-ny}. \quad (1.21)$$

Widzimy zatem, że jest to element wykładniczej rodziny dyspersji. W tym przypadku kanoniczna funkcja związku to funkcja logit. Pierwszy moment wyraża się przez

$$p = \mathbb{E}_\theta[Y] = \kappa'(\theta) = \frac{e^\theta}{1 + e^\theta}. \quad (1.22)$$

Wariancja wyraża się przez

$$\text{Var}_\theta(Y) = \frac{1}{n} \kappa''(\theta) = \frac{1}{n} \frac{e^\theta}{(1 + e^\theta)^2} = \frac{1}{n} p(1 - p). \quad (1.23)$$

Funkcja wariancji wyraża się przez

$$V(\mu) = \mu(1 - \mu). \quad (1.24)$$

Zmienna losowa z rozkładu dwumianowego uzyskana jest poprzez $X = nY \sim \text{Binom}(n, p)$.

Inne przykładowe rozkłady z wykładniczej rodziny rozkładów dyspersyjnych to: rozkład Poissona, rozkład gamma, rozkład Walda.

1.7 Uogólniony model liniowy

W niniejszym podrozdziale omawiamy koncepcję uogólnionego modelu liniowego, który stoi u podstaw formalizmu sztucznych sieci neuronowych. Niniejszy podrozdział powstał na podstawie [31, rozdz. 5].

Rozpoczniemy od definicji cech $\mathbf{x} \in \mathcal{X}$. Cechy nazywane są także zmiennymi objaśniającymi, zmiennymi niezależnymi lub regresorami. Założymy, że cechy $\mathbf{x} = (x_0, x_1, \dots, x_q)^\top$ zawierają pierwszy element $x_0 = 1$ oraz że wybieramy przestrzeń cech $\mathcal{X} \subset \{1\} \times \mathbb{R}^q$. Element 1 nazywany też wyrazem wolnym (ang. *intercept*) lub członem obciążenia (ang. *bias component*).

Załączmy, że mamy niezależne zmienne losowe Y_1, \dots, Y_n , które są opisywane przez jeden z rozkładów z rodziny rozkładów wykładniczych z dyspersją. Oznacza to, że wszystkie Y_i są niezależne i mają gęstości względem σ -skończonej miary ν na \mathbb{R} dane przez

$$Y_i \sim f(y_i; \theta_i, v_i/\varphi) = \exp\left(\frac{y_i \theta_i - \kappa(\theta_i)}{\varphi/v_i} + a(y_i; v_i/\varphi)\right), \quad (1.25)$$

dla $1 \leq i \leq n$, kanonicznych parametrów $\theta_i \in \Theta$, wag $v_i > 0$ i parametru dyspersji $\varphi > 0$. Zwróćmy uwagę, że każda zmienna losowa Y_i może mieć własny parametr kanoniczny θ_i .

W modelowaniu statystycznym istnieje pojęcie modelu nasyconego (pełnego, ang. *saturated model*), w którym każda obserwacja Y_i ma swój własny parametr kanoniczny θ_i . Ogólnie, jeśli mamy n obserwacji $\mathbf{Y} = (Y_1, \dots, Y_n)^\top$, możemy oszacować co najwyżej n parametrów kanonicznych. Drugi skrajny przypadek to model jednorodny, co oznacza, że $\theta_i = \theta \in \Theta$ dla wszystkich $1 \leq i \leq n$. W drugim przypadku mamy dokładnie jeden parametr kanoniczny do oszacowania. Nazywamy ten model modelem pustym (ang. *null model*), modelem z wyrazem wolnym lub modelem jednorodnym, ponieważ zakładamy, że wszystkie składowe \mathbf{Y} podlegają tej samej regule wyrażonej w jednym wspólnym parametrze kanonicznym θ .

Zarówno model nasycony, jak i model pusty mogą zachowywać się bardzo słabo w predykcji nowych obserwacji. Zazwyczaj model nasycony w pełni odzwierciedla dane \mathbf{Y} , w tym także część szumową (składnik losowy), dlatego nie jest przydatny do predykcji. Mówimy również, że ten model (w próbce) dopasowuje się nadmiernie (ang. *overfitting*, zob. podrozdział 1.5) do danych \mathbf{Y} i nie generalizuje (poza próbki) do nowych danych. Model pusty często ma słabą wydajność predykcyjną, ponieważ jeśli dane mają efekty systematyczne, nie można ich uchwycić za pomocą modelu pustego. Uogólnione modele liniowe starają się znaleźć dobrą równowagę między tymi dwoma skrajnymi przypadkami i próbują wyodrębnić (tylko) efekty systematyczne z zaszumionych danych Y . W tym celu modelujemy parametry kanoniczne θ_i jako funkcję $\theta_i(\mathbf{x})$ o małym wymiarze zmiennych objaśniających, które uchwytują efekty systematyczne w danych.

W naszych rozważaniach zakładamy, że każda obserwacja Y_i jest stwarzyszona z cechami (zmiennymi objaśniającymi) \mathbf{x}_i , które należą do pewnej przestrzeni cech \mathcal{X} . Cechy \mathbf{x}_i w założeniu mają opisywać systematyczne efekty dla obserwacji Y_i . Możemy myśleć o tym, jak poszczególne cechy \mathbf{x}_i „systematycznie” wpływają na zmienną odpowiedzi Y_i , czyli jakie zmiany w cechach mają wpływ na przewidywaną wartość zmiennej odpowiedzi.

Podsumowując, zakładamy, że mamy odpowiednią funkcję $\theta: \mathcal{X} \rightarrow \Theta$, nazywaną funkcją regresji, która zależy od cech, czyli $\theta(\mathbf{x})$. Funkcja ta pozwala nam odpowiednio opisywać obserwacje przez

$$Y_i \sim f(y_i; \theta_i = \theta(\mathbf{x}), v_i/\varphi) = \exp\left(\frac{y_i \theta(\mathbf{x}) - \kappa(\theta(\mathbf{x}))}{\varphi/v_i} + a(y_i; v_i/\varphi)\right), \quad (1.26)$$

dla $1 \leq i \leq n$. W rezultacie otrzymujemy pierwszy moment Y_i :

$$\mu_i = \mu(\mathbf{x}_i) = \mathbb{E}_{\theta(\mathbf{x}_i)}[Y_i] = \kappa'(\theta(\mathbf{x}_i)). \quad (1.27)$$

Stąd zakładamy, że funkcja regresji $\theta: \mathcal{X} \rightarrow \Theta$ opisuje systematyczne efekty (różnice) pomiędzy zmiennymi Y_1, \dots, Y_n wyrażone przez średnie μ_i dla cech $\mathbf{x}_1, \dots, \mathbf{x}_n$. W uogólnionym modelu liniowym funkcja regresji jest liniowa, co uzasadnia terminologię liniowości modelu uogólnionego.

Przechodzimy do definicji uogólnionego modelu liniowego.

Definicja 1.16 (Uogólniony model liniowy) Załączmy istnienie parametrów regresji $\beta \in \mathbb{R}^{q+1}$ i istnienie ściśle monotonicznej i gładkiej funkcji $g: \mathcal{M} \rightarrow \mathbb{R}$. Wtedy *uogólnionym modelem liniowym* (ang. *generalized linear model*) nazywamy (opuszczamy indeks dolny przy \mathbf{x}):

$$g(\mu(\mathbf{x})) = g(\mathbb{E}_{\theta(\mathbf{x})}[Y]) = \eta(\mathbf{x}) = \langle \boldsymbol{\beta}, \mathbf{x} \rangle = \beta_0 + \sum_{j=1}^q \beta_j x_j. \quad (1.28)$$

Symbol $\langle \cdot, \cdot \rangle$ oznacza iloczyn skalarny w przestrzeni euklidesowej \mathbb{R}^{q+1} i wykorzystany jest tu jako skrót zapisu sumy. $\theta(\mathbf{x}) = h(\mu(\mathbf{x}))$ to parametr kanoniczny dla kanonicznej funkcji związku

h , $\eta(\mathbf{x})$ jest nazywany predyktorem liniowym. Funkcja g określana jest mianem funkcji związku (ang. *link function*).

Możemy zauważyć, że (1.28) to (1.27) z zastosowaniem funkcji związku g . Po zastosowaniu funkcji związku g efekty systematyczne zmiennej losowej Y z cechami \mathbf{x} mogą być opisane przez predyktor liniowy $\eta(\mathbf{x}) = \langle \boldsymbol{\beta}, \mathbf{x} \rangle$. To daje nam szczególną postać dla (1.27) oraz oznacza, że zmienne losowe Y_1, \dots, Y_n dzielą wspólne parametry regresji $\boldsymbol{\beta} \in \mathbb{R}^{q+1}$. Dodatkowo warto zaznaczyć, że funkcja związku g może różnić się od kanonicznej funkcji związku h .

Podczas rozważań nad uogólnionym modelem liniowym należy odpowiedzieć na kilka zasadniczych kwestii.

Po pierwsze należy wybrać odpowiedni rozkład prawdopodobieństwa z rodziny rozkładów wykładniczych z dyspersją. W tym celu powinniśmy spróbować zrozumieć właściwości danych Y , zanim dokonamy tego wyboru. Na przykład: czy mamy zadanie klasyfikacji? Albo: czy mamy obserwacje ciągłe? Czasami dane Y są najpierw przekształcone, zanim zostaną zamodelowane za pomocą rozkładu prawdopodobieństwa. Popularnym przekształceniem jest logarytm dla obserwacji dodatnich. Po tym przekształceniu można wybrać element rodziny rozkładów wykładniczych z dyspersją do modelowania $\log(Y)$. Jeśli przykładowo wybierzemy rozkład Gaussa dla $\log(Y)$, to Y będzie miało rozkład log-normalny, natomiast jeśli wybierzemy rozkład wykładniczy dla $\log(Y)$, to Y będzie miało rozkład Pareto; wynika to z własności rozkładów prawdopodobieństwa, dowody tych faktów można znaleźć w [31]. Następnie można zamodelować przekształcone dane za pomocą uogólnionego modelu liniowego. Często zapewnia to bardzo dokładne modele, na przykład w skali logarytmicznej dla przekształconych danych.

Po drugie, ważnym zagadnieniem jest wybór funkcji związku g . Jest ona ściśle związana z zadaniem, jakie próbujemy rozwiązać, gdyż definiuje relacje między wartością oczekiwanaą zmiennej Y a cechami \mathbf{x} . Ponieważ chcemy modelować wartość oczekiwanaą zmiennej objaśnianej Y , to należy взять funkcję odwrotną funkcji związku dla poznania poszukiwanej zależności. Funkcja związku musi być odpowiednio dobrana do założonego rozkładu Y . Przykładowo, jeżeli zakładamy, że zmienna Y ma rozkład normalny, to popularnym wyborem dla funkcji g będzie identycznościowa funkcja związku, tzn.

$$g(\mu(\mathbf{x})) = \text{id}(\mu(\mathbf{x})) = \langle \boldsymbol{\beta}, \mathbf{x} \rangle. \quad (1.29)$$

Funkcja odwrotna do funkcji identycznościowej to funkcja identycznościowa, zatem

$$\text{id}(\mu(\mathbf{x})) = \mu(\mathbf{x}) = \text{id}(\langle \boldsymbol{\beta}, \mathbf{x} \rangle) = \langle \boldsymbol{\beta}, \mathbf{x} \rangle, \quad (1.30)$$

co w rezultacie daje nam zadanie regresji wielorakiej i jest to klasyczny przykład zadania regresji. Natomiast w przypadku założenia rozkładu Bernoulliego dla Y możemy przyjąć funkcję logitową jako funkcję związku, tzn.

$$g(\mu(\mathbf{x})) = \ln \left(\frac{\mu(\mathbf{x})}{1 - \mu(\mathbf{x})} \right) = \langle \boldsymbol{\beta}, \mathbf{x} \rangle. \quad (1.31)$$

Zatem biorąc funkcję odwrotną, mamy

$$\mu(\mathbf{x}) = \frac{\exp(\langle \boldsymbol{\beta}, \mathbf{x} \rangle)}{1 - \exp(\langle \boldsymbol{\beta}, \mathbf{x} \rangle)}. \quad (1.32)$$

Jest to model regresji logistycznej, a więc klasyczny przykład modelu dla zadania klasyfikacji.

Widzimy zatem, że uogólniony model liniowy może służyć zarówno w zadaniach klasyfikacji, jak i regresji oraz stanowi podstawę dla wielu klasycznych modeli uczenia maszynowego.

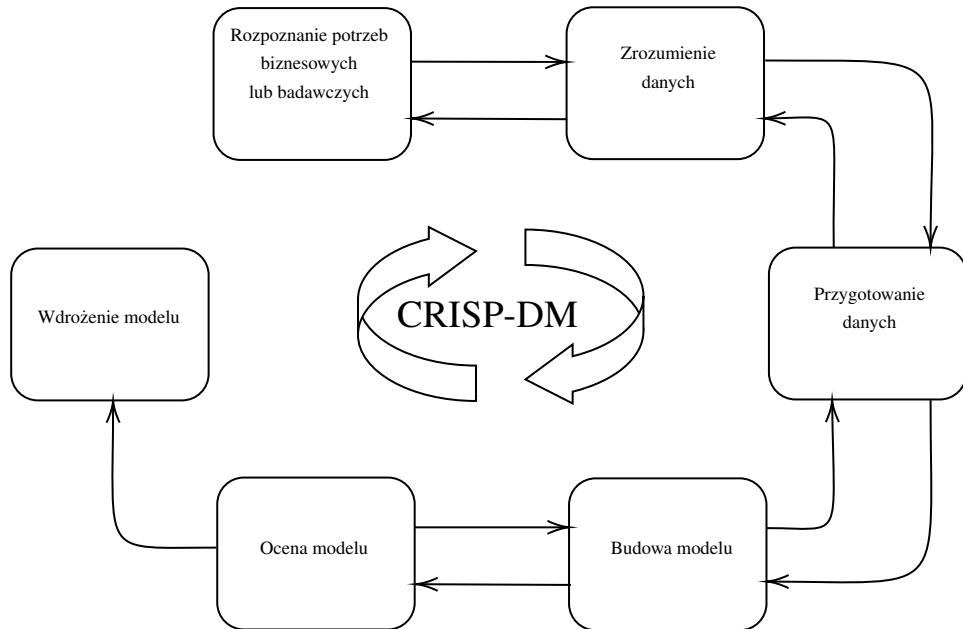
1.8 Standardowy proces eksploracji danych CRISP-DM

W tym podrozdziale prezentujemy efektywny sposób prowadzenia projektu związanego z eksploracją danych, zgodnie z którym postępować będziemy w części eksperimentalnej pracy (rozdz. 4). Niniejszy podrozdział powstał na podstawie [16, podrozdz. 1.4].

Projekty związane z eksploracją danych rodzą wiele problemów, z którymi trzeba się mierzyć podczas ich realizacji. Jedną z trudności są dane niskiej jakości, a więc dane niekompletne lub zaszumione. Niepowtarzalność projektów jest kolejną trudnością, ponieważ praktycznie niemożliwe jest uzyskanie satysfakcyjnych rezultatów z wykorzystaniem dokładnie tych samych narzędzi eksploracji danych dla dwóch różnych projektów. W efekcie powstaje mnogość możliwych rozwiązań, która czyni dziedzinę eksploracji danych bardzo wymagającą pod względem wiedzy, umiejętności i doświadczenia, aby móc skutecznie i efektywnie wykorzystać potencjał eksploracji danych do rozwiązywania rzeczywistych problemów biznesowych i naukowych.

Pomimo znaczących sukcesów na polu automatyzacji procesów eksploracji danych, wciąż istotna jest rola analityka danych, który nadzoruje poszczególne działania, wprowadza zmiany, interpretuje ich wyniki oraz ma możliwość rozwiązywania zaistniałych trudności. Podejmowano próby

usystematyzowania wykonywanych działań analityków w celu optymalizacji efektów pracy nad danymi. Rozwiązaniem, które obecnie znajduje powszechnie zastosowanie, jest proces CRISP-DM (ang. *The Cross-Industry Standard Process for Data Mining*). CRISP-DM został w całości po raz pierwszy opublikowany w [30].



Rysunek 1.1: Schemat procesu CRISP-DM. Opracowanie własne na podstawie [16, rys. 1.1].

CRISP-DM to iteracyjny proces, który określa ogólną strategię, jaką można przyjąć podczas realizacji projektów eksploracji danych w celu maksymalizacji efektywności. Na rys. 1.1 przedstawiono ogólny schemat procesu CRISP-DM. Składa się z sześciu etapów:

1. Rozpoznanie potrzeby biznesowej lub badawczej.
 - (a) Sformułowanie celów, wymagań i ograniczeń projektu w kontekście problemu badawczego lub biznesowego.
 - (b) Przetłumaczenie celów projektu, z uwzględnieniem wymagań i ograniczeń, na zadania w kontekście eksploracji danych.
 - (c) Przygotowanie wstępnej strategii umożliwiającej osiągnięcie celów projektowych.
2. Zrozumienie danych.
 - (a) Zebranie danych.
 - (b) Zapoznanie się z danymi i zrozumienie ich istoty przy wykorzystaniu narzędzi eksploracyjnej analizy danych.
 - (c) Ocenienie jakości danych.
 - (d) Opcjonalne zawężenie zbioru danych do interesujących nas podzbiorów.
3. Przygotowanie danych.
 - (a) Wybranie odpowiednich przypadków i zmiennych istotnych dla problemu.
 - (b) Opcjonalne transformacje zmiennych.
 - (c) Oczyszczenie danych do stanu używalnego w kontekście budowy modeli.
4. Budowa modelu.
 - (a) Wybranie oraz zastosowanie odpowiednich technik modelowania danych.
 - (b) Optymalizacja modeli poprzez kalibrację parametrów.

5. Ocena modelu pod kątem jakości i efektywności.
 - (a) Ocenienie, czy stworzone modele spełniają cele postawione w pierwszym etapie.
 - (b) Upewnienie się, czy nie pominięto ważnych elementów w kontekście problemu projektowego.
 - (c) Podjęcie decyzji, czy modele mogą zostać wykorzystane.
6. Wdrożenie modelu.
 - (a) Wykorzystanie modeli do rozwiązywania rzeczywistych problemów.

Podczas realizacji kolejnych kroków procesu mogą pojawić się nowe wątpliwości, które nie występowały na wcześniejszych etapach. Stąd istnieje możliwość powrotu do wcześniejszego etapu w celu uwzględnienia nowo powstałych problemów i ulepszenia istniejących rozwiązań. Z tego powodu na rys. 1.1 każdy z etapów (poza etapem 6, który jest końcem procesu) ma połączenie wsteczne z poprzednim elementem.

Rozwiązań rzeczywistych problemów biznesowych i badawczych związanych z eksploracją danych mogą zostać osiągnięte przez przełożenie wiedzy i doświadczenia z przeszłych projektów na grunt nowych wyzwań. Ta właściwość procesu CRISP-DM symbolizowana jest na rys. 1.1 przez strzałki tworzące okrąg wewnątrz grafu.

W niniejszej pracy magisterskiej, w części eksperimentalnej pracy (rozdz. 4), wykonano eksplorację danych zgodnie z opisanymi etapami CRISP-DM.

Rozdział 2

Wprowadzenie do teorii sztucznych sieci neuronowych i uczenia głębokiego

W tym rozdziale omówimy podstawy teorii sztucznych sieci neuronowych i uczenia głębokiego. Nasze rozważania rozpocznemy od ogólnej koncepcji jednokierunkowej sztucznej sieci neuronowej, a następnie przedstawimy najprostszy przykład takiej sieci – model perceptronu i jego rozwinięcie do perceptronu wielowarstwowego. Sformalizujemy nasze rozważania i przedstawimy zwięzły opis matematyczny jednokierunkowej sztucznej sieci neuronowej. Na koniec przedstawimy metody treningu sztucznych sieci neuronowych wykorzystywanych także w bardziej zaawansowanych modelach, takich jak sieci splotowe.

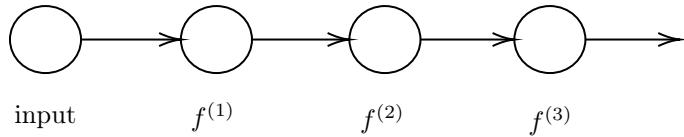
2.1 Jednokierunkowe sztuczne sieci neuronowe

W tym podrozdziale omawiamy ogólną koncepcję sztucznej sieci neuronowej. Niniejszy podrozdział powstał na podstawie [9, rozdz. 6].

Jednokierunkowe sztuczne sieci neuronowe (ang. *feedforward artificial neural networks*) to podstawowe modele w dziedzinie uczenia głębokiego. Zasadniczym celem jednokierunkowej sztucznej sieci neuronowej jest aproksymacja pewnej, na ogólnie nieznanej funkcji $\hat{f}: \mathcal{X} \rightarrow \mathcal{Y}$. Funkcja \hat{f} opisuje rzeczywistą relację pomiędzy danymi wejściowymi $\mathbf{X} \subset \mathcal{X}$, a wyjściowym wektorem wartości $\mathbf{y} \subset \mathcal{Y}$. Dla jednokierunkowej sztucznej sieci neuronowej możemy rozważać zadania klasyfikacji i zadania regresji, zatem zbiór \mathbf{y} może mieć różnych charakter w zależności od typu zadania. Niezależnie od zadania, dla danej próbki (obserwacji) mamy $\hat{f}_{\theta}(\mathbf{x}) = \mathbf{y}$, gdzie $\mathbf{x} \in \mathbf{X}$, $\mathbf{y} \in \mathcal{Y}$ i $\theta \in \Theta$. θ to wektor parametrów sieci z pewnej przestrzeni parametrów Θ . Jednokierunkowa sztuczna sieć neuronowa w dążeniu do znalezienia \hat{f} definiuje odwzorowanie $f_{\theta}(\mathbf{X}) = \hat{\mathbf{y}} \approx \mathbf{y}$ oraz, poprzez trening sieci, poszukuje optymalnych wartości parametrów θ , które skutkują najbardziej odpowiednią aproksymacją funkcji \hat{f} . W dalszej części możemy pomijać argument θ przy zapisie funkcji.

Jednokierunkowa sztuczna sieć neuronowa nazywana jest „siecią” z uwagi na fakt, że na ogół powstaje ona w wyniku złożenia ze sobą kilku różnych funkcji. Sposób złożenia ze sobą funkcji i ich wzajemne relacje są często wyrażane w postaci acyklicznego grafu skierowanego. Przykładowo, jednokierunkowa sztuczna sieć neuronowa może składać się z trzech funkcji $f^{(1)}, f^{(2)}, f^{(3)}$ złożonych ze sobą w formie łańcucha w taki sposób, że $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. Graf, który odpowiada takiemu określeniu f , przedstawiono na rys. 2.1. Taka struktura łańcucha to jedna z najprostszych struktur jednokierunkowych sztucznych sieci neuronowych. Na początku mamy warstwę wejściową (ang. *input layer*) opisywaną często jako *input*, następuje tu przekazanie danych do sieci. $f^{(1)}$ nazywamy pierwszą warstwą ukrytą, $f^{(2)}$ nazywamy drugą warstwą ukrytą itd. Ostatnią warstwę sieci nazywamy warstwą wyjściową sieci (ang. *output layer*, tu $f^{(3)}$). Całkowita długość łańcucha określa głębokość (ang. *depth*) modelu, stąd właśnie wywodzi się nazwa „uczenie głębokie”. Wartości wyjściowe dla poszczególnych warstw (poza warstwą wyjściową) nie są znane, oznacza to, że są tylko używane wewnętrznie i nie są odczytywane z zewnątrz podczas stosowania sieci neuronowej. Z tego powodu wszystkie warstwy poza pierwszą i ostatnią nazywane są zbiorczo warstwami ukrytymi sieci (ang. *hidden layers*).

Określenie „jednokierunkowe” bierze swą genezę z mechanizmu przepływu danych w modelu. Dane trafiają do warstwy wejściowej, przechodzą przez warstwy ukryte, aż do warstwy wyjściowej. Na każdym z tych etapów na danych wykonywane są pewne przekształcenia (omówione



Rysunek 2.1: Przykład prostej architektury jednokierunkowej sztucznej sieci neuronowej składającej się z trzech warstw kolejno: *input* – warstwa wejściowa, $f^{(1)}$ – pierwsza warstwa ukryta, $f^{(2)}$ – druga warstwa ukryta, $f^{(3)}$ – warstwa wyjściowa. W tym przypadku każda z warstw składa się z jednej jednostki (neuronu), w ogólności warstwy mogą się składać z wielu neuronów (zob. rys. 2.3 lub rys. 2.5). Schematy architektury sieci mogą się różnić w zależności od źródła. Opracowanie własne.

bardziej szczegółowo na przykładzie perceptronu w podrozdziale 2.2). Na gruncie grafowej wizualizacji jednokierunkowej sztucznej sieci neuronowej możemy powiedzieć, że kolejne wierzchołki grafów reprezentujące neurony z kolejnych warstw sieci nie mają połączenia do neuronów pochodzących z wcześniejszych wierzchołków).

Jednokierunkowe sztuczne sieci nazywane są „neuronowymi” dlatego, że inspiracja do powstania tego modelu została zaczerpnięta ze struktury sieci komórek nerwowych czyli neuronów. Na podstawie budowy biologicznego neuronu został opracowany prosty model sztucznego neuronu (ang. *artificial neuron*). Składa się z co najmniej jednego wejścia i jednego wyjścia. Aktywnienie wyjścia sztucznego neuronu następuje, gdy aktywna jest określona liczba wejść. Sztuczny neuron, inaczej nazywany jednostką (ang. *unit*), otrzymując pewne wartości na wejściu, oblicza pewną wartość wyjściową. Zwyczajowo mówimy, że do sztucznego neuronu przesyłane są sygnały wejściowe, a wysyłany jest sygnał wyjściowy. Na ogólnie każdy warstwa ukryta sieci ma wartość wektorową. Rozpatrując funkcję $f^{(i)}$, odpowiadającą i -tej warstwie ukrytej, jako obiekt składający się z wielu jednostek o charakterze funkcjonalu, można powiedzieć, że każdy neuron za pomocą tzw. funkcji aktywacji (ang. *activation function*) przyjmuje wektor sygnałów wejściowych i wysyła wartość sygnału wyjściowego.

2.2 Perceptron

W niniejszym podrozdziale prezentujemy przykład prostego modelu jednokierunkowej sztucznej sieci neuronowej – perceptronu. Podrozdział powstał na podstawie [7, rozdz. 10].

Jednym z najprostszych przykładów jednokierunkowej sztucznej sieci neuronowej jest model perceptronu, na podstawie którego omówimy kolejne istotne idee uczenia głębokiego.

Perceptron to model jednowarstwowej sztucznej sieci neuronowej opracowany w 1962 roku i opublikowany w [23]. Składa się ze sztucznego neuronu nazywanego progową jednostką logiczną (ang. *Threshold Logic Unit* – TLU). TLU jako wejście przyjmuje liczby oraz każde połączenie ma przypisaną pewną liczbę, którą nazywamy wagą (ang. *weight*). Otrzymując wektor sygnałów wejściowych TLU oblicza ważoną sumę sygnałów wejściowych (ważoną zgodnie z wagami połączeń neuronów).

W dalszej kolejności do wyznaczonej ważonej sumy sygnałów zastosowana jest funkcja, która w modelu perceptronu nazywana jest funkcją aktywacji. W literaturze zostało zaproponowanych wiele funkcji aktywacji dla perceptronów, jednak najczęściej wykorzystywana jest funkcja skokowa przedstawiona w tabeli 2.1 na stronie 24.

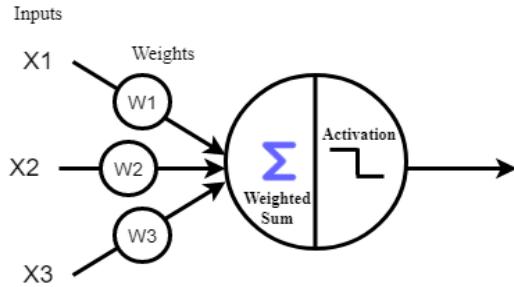
Podsumowując, mając zestaw danych wejściowych x_i i zestaw wag połączeń w_i dla $1 \leq i \leq N \in \mathbb{N}$, a także funkcję aktywacji $\phi: \mathbb{R} \rightarrow \mathbb{R}$, operacja w TLU wyraża się wzorem:

$$\text{TLU} = \phi \left(\sum_{i=1}^N w_i x_i \right). \quad (2.1)$$

Operację wykonywaną w TLU zilustrowano na rys. 2.2

TLU można wykorzystać w prostych zadaniach klasyfikacji binarnej. Określamy wówczas pewną wartość progową, a następnie klasyfikację danych wejściowych przeprowadzamy w ten sposób, że jeżeli wartość ważonej sumy sygnałów wejściowych z zastosowaniem funkcji aktywacji przekracza ustalony próg, to obserwację zaklasyfikujemy do jednej klasy, a jeżeli nie przekracza – do drugiej. Jest to przykład modelu liniowego.

Perceptron to sztuczna sieć neuronowa składająca się z jednej warstwy TLU, w której każdy z neuronów połączony jest ze wszystkimi neuronami z warstwy wejściowej. Warstwa, której neurony mają taką własność, nazywana jest warstwą w pełni połączoną lub warstwą gęstą. Na ogólnie do warstwy wejściowej dodawany jest jeszcze tak zwany neuron obciążeniowy (ang. *bias neuron*), który zawsze wysyła wartość 1.



Rysunek 2.2: Operacja wykonywana w TLU. W tym przypadku mamy 3 wejścia oraz 3 wagi połączeń. Źródło: [25].

Uczenie perceptronu i w ogólności sztucznej sieci neuronowej polega na znalezieniu odpowiednich wartości wag połączeń. Algorytm treningu TLU jest odmianą reguły Hebb'a¹ tzn. zwiększa się wagi tych połączeń, które pozwalają zmniejszyć wartość błędu popełnianego przez sieć podczas prognozy. Najpierw wyznaczane są prognozy dla każdego przykładu uczącego; potem dla każdego neuronu wyjściowego, w którym pojawił się wynik nieprawidłowy, zwiększa się wagi tych połączeń z wejściami, które przyczyniają się do prognozy właściwej.

Metoda uczenia perceptronu daje nam gwarancję znalezienia optymalnych wag (jest zbieżna) jeżeli próbki uczące są liniowo rozdzielne (wzorce wykorzystywane do uczenia perceptronu tworzą dwie separowalne klasy).

Istnieją zadania (nawet trywialne), których model perceptronu nie jest w stanie rozwiązać, np. zagadnienie klasyfikacyjne alternatywy rozłącznej. Z tego powodu rozwinięto model perceptronu do perceptronu wielowarstwowego.

2.3 Perceptron wielowarstwowy

W tym podrozdziale przedstawiamy rozwinięcie modelu perceptronu do modelu z wieloma warstwami ukrytymi. Niniejszy podrozdział powstał na podstawie [7, rozdz. 10].

W związku z ograniczeniami perceptronu zaczęto rozwijać początkowe modele. W ten sposób powstał model perceptronu wielowarstwowego (ang. *Multi-Layer Perceptron*) opublikowany w [20]. Ma on zdolność rozwiązywania zadań niemożliwych do rozwiązania dla zwykłego perceptronu. Model perceptronu wielowarstwowego stanowi podstawę dla wielu zaawansowanych modeli uczenia głębokiego takich jak splotowe sztuczne sieci neuronowe będące trzonem dziedziny widzenia komputerowego.

Na architekturę perceptronu wielowarstwowego składa się warstwa wejściowa, jedna lub więcej warstw ukrytych jednostek TLU oraz ostatnia warstwa wyjściowa jednostek TLU. Zwyczajowo te warstwy, które znajdują się bliżej warstwy wejściowej, nazywane są warstwami dolnymi (ang. *lower layer*), a te bliżej warstwy wyjściowej określone są mianem warstw górnych (ang. *upper layers*). Na rys. 2.3 przedstawiono przykładową architekturę wielowarstwowego perceptronu.

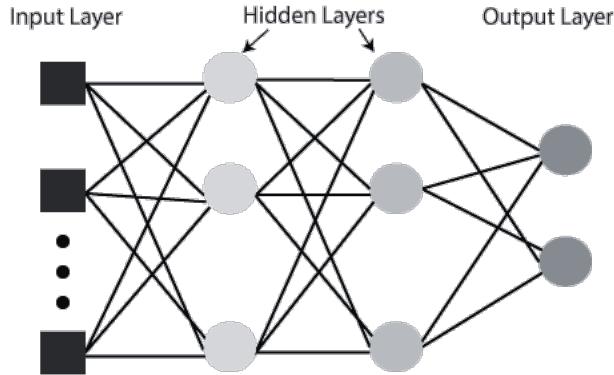
Uczenie perceptronu wielowarstwowego jest zagadniением bardziej skomplikowanym niż uczenie zwykłego perceptronu. Z uwagi na tę trudność powstała potrzeba stworzenia nowego algorytmu, którym jest algorytm wstecznej propagacji błędu opisany w podrozdziale 2.6. Nowy algorytm treningu wymaga także dodatkowej zmiany w architekturze sieci perceptronu wielowarstwowego. Z potrzeby użycia pochodnych funkcji aktywacji w algorytmie propagacji wstecznej koniecznym jest wybór funkcji aktywacji, które są różniczkowalne (funkcje skokowe w TLU takie nie są).

2.4 Warstwa porzucenia

W tym podrozdziale opisujemy element zmniejszający ryzyko przetrenowania modelu – warstwę porzucenia.

Warstwa porzucenia (ang. *dropout layer*) to element regularyzacji sieci (zob. podrozdział 1.5).

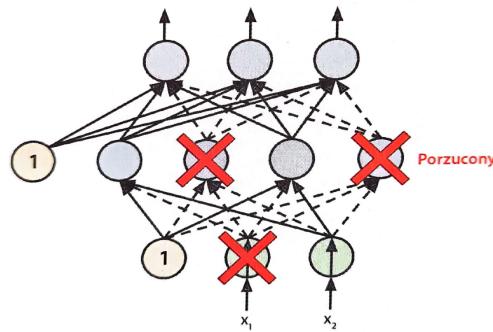
¹W neurobiologii reguła Hebb'a (lub uczenie hebbowskie) to koncepcja, która mówi, że jeżeli dwie komórki nerwowe oddziałują na siebie, to połączenie między nimi staje się silniejsze.



Rysunek 2.3: Przykład wielowarstwowego perceptronu. Najpierw mamy warstwę wejściową (*Input Layer*), gdzie dane są zaczynane do modelu. Potem następują warstwy ukryte (*Hidden Layers*), w tym przypadku są dwie. Na koniec mamy warstwę wyjściową (*Output Layer*), która umożliwia wykonanie predykcji. Źródło: [10].

Ta warstwa działa poprzez losowe wyłączanie (zerowanie) pewnej liczby neuronów podczas treningu. To oznacza, że w każdym kroku treningu niektóre jednostki są wyłączone, co powoduje, że model staje się bardziej odporny na przetrenowanie na treningowym zestawie danych. W każdym kroku treningu, warstwa porzucenia losowo wybiera pewien odsetek jednostek neuronowych do wyłączenia. Ten odsetek jest określany jako współczynnik porzucenia, zazwyczaj ustawiany na wartość pomiędzy 0,2 a 0,5. Wybrane jednostki są ustawiane na zero w danym kroku treningu. To oznacza, że nie przekazują one żadnej informacji do kolejnych warstw sieci w tym konkretnym kroku.

Warstwa porzucenia jest szczególnie przydatna w przypadku dużych i złożonych modeli lub w sytuacjach, gdzie dostępne dane treningowe są ograniczone. Jednak należy zachować umiar w ustawianiu współczynnika porzucenia, aby nie wprowadzić zbyt dużego zakłócenia w procesie uczenia. Ostatecznie, w fazie testowania (poza treningiem), warstwa porzucenia jest wyłączona, a wszystkie neurony działają normalnie, co pozwala na uzyskanie dokładniejszych prognoz.



Rysunek 2.4: Warstwa porzucenia. Część jednostek jest wyłączała (wysyłają wartość 0). Źródło: [7].

2.5 Jednokierunkowa sieć neuronowa w ujęciu formalnym

W tym podrozdziale przedstawiamy formalizm matematyczny stojący za jednokierunkowymi sztuczными sieciami neuronowymi. Niniejszy podrozdział powstał w oparciu o [31, rozdz. 7].

W uogólnionym modelu liniowym modelujemy wartość oczekiwana (średnia) zmiennej objaśnianej Y , mając cechy \mathbf{x} , za pomocą równania:

$$\mu(\mathbf{x}) = \mathbb{E}_{\theta(\mathbf{x})}[Y] = g^{-1}(\langle \boldsymbol{\beta}, \mathbf{x} \rangle). \quad (2.2)$$

Poszczególne obiekty w równaniu (2.2) wyjaśniono w podrozdziale 1.7. Najważniejszym założeniem równania (2.2) jest to, że uogólniony model liniowy dostarcza racjonalnego i funkcjonalnego opisu wartości oczekiwanej $\mathbb{E}_{\theta(\mathbf{x})}[Y]$.

Warstwy sztucznej sieci neuronowej traktujemy jako skończony ciąg funkcji, oznaczmy je tu jako $(\mathbf{z}^{(m)})_{1 \leq m \leq d}$, określone jako

$$\mathbf{z}^{(m)}: \{1\} \times \mathbb{R}^{q_{m-1}} \rightarrow \{1\} \times \mathbb{R}^{q_m}, \quad (2.3)$$

gdzie $q_m \in \mathbb{N}$ dla $1 \leq m \leq d$ to wymiary, a wymiar $q_0 = q$ to warstwa wejściowa dla wektora cech wejściowych $\mathbf{x} \in \mathcal{X} \subset \{1\} \times \mathbb{R}^q$.

Każda warstwa tworzy reprezentację cech (ang. *representation of the features*), to znaczy, że po m -tej warstwie mamy q_m -wymiarową reprezentację cech wejściowych $\mathbf{x} \in \mathcal{X}$

$$\mathbf{z}^{(m:1)}(\mathbf{x}) := (\mathbf{z}^{(m)} \circ \dots \circ \mathbf{z}^{(1)})(\mathbf{x}) \in \{1\} \times \mathbb{R}^{q_m}. \quad (2.4)$$

Pierwszy komponent zawsze jest równy 1, z tego powodu reprezentację $\mathbf{z}^{(m:1)}(\mathbf{x}) \in \{1\} \times \mathbb{R}^{q_m}$ określamy jako q_m -wymiarową.

W teorii uczenia głębokiego zakładamy, że mamy $d \in \mathbb{N}$ odpowiednich transformacji (czyli warstw) $\mathbf{z}^{(m)}$, $1 \leq m \leq d$, takich, że $\mathbf{z}^{(d:1)}(\mathbf{x})$ zapewnia odpowiednią q_d -wymiarową reprezentację cech wejściowych $\mathbf{x} \in \mathcal{X}$, wówczas wracając do uogólnionego modelu liniowego mamy

$$\mu(\mathbf{x}) = \mathbb{E}_{\theta(\mathbf{x})}[Y] = g^{-1} \left(\langle \boldsymbol{\beta}, \mathbf{z}^{(d:1)}(\mathbf{x}) \rangle \right), \quad (2.5)$$

z funkcją związku $g: \mathcal{M} \rightarrow \mathbb{R}$ oraz parametrem $\boldsymbol{\beta} \in \mathbb{R}^{q_d+1}$. Taka architektura modelu tworzy nam jednokierunkową sztuczną sieć neuronową o wymiarze $d \in \mathbb{N}$, gdyż informacja w postaci zestawu cech \mathbf{x} jest przetwarzana w acyklicznym grafie skierowanym przez d warstw $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(d)}$ zanim trafi ostatecznie do uogólnionego modelu liniowego.

Każda warstwa ukryta $\mathbf{z}^{(m)}$ zawiera parametry, które tożsame są z wagami połączeń sieci i członem obciążenia. Uczenie głębokie dopasowuje parametry warstw, a także parametry $\boldsymbol{\beta}$ uogólnionego modelu liniowego na wyjściu, do dostępnych danych treningowych $\mathcal{L} \subset \mathcal{X}$ uzyskując optymalny model predykcyjny dla danych testowych $\mathcal{T} \subset \mathcal{X}$. Wówczas wyuczony model powinien optymalnie generalizować nieznane dane. Proces optymalnego uczenia reprezentacji jest również częścią procedury dopasowania modelu. Nowoczesne dopasowanie modeli w uczeniu głębokim wykorzystuje warianty algorytmu gradientu prostego, który omówimy w podrozdziale 2.6.

Jednokierunkowe sztuczne sieci neuronowe to funkcje typu (2.5), gdzie każdy neuron $z_j^{(m)}$, $1 \leq j \leq q_m$, warstwy $\mathbf{z}^{(m)} = (1, z_1^{(m)}, \dots, z_{q_m}^{(m)})^\top$, $1 \leq m \leq d$, ma strukturę uogólnionego modelu liniowego. Pierwszy komponent $z_0^{(m)} = 1$ zawsze pełni rolę członu obciążenia (lub inaczej wyrazu wolnego) i nie wymaga modelowania.

Pierwszym wyborem w konstrukcji jednokierunkowej sztucznej sieci neuronowej jest wybór funkcji aktywacji $\phi: \mathbb{R} \rightarrow \mathbb{R}$, która pełni rolę odwrotnej funkcji związku. W celu uczenia nieliniiowych reprezentacji funkcja aktywacji powinna być nielinowa. Najpopularniejsze funkcje aktywacji przedstawiono w tabeli 2.1.

Tabela 2.1: Wybrane funkcje aktywacji neuronu stosowane w sieciach.

Nazwa	Wzór	Pochodna
Logistyczna (sigmoidalna)	$\phi(x) = (1 + \exp(-x))^{-1}$	$\phi' = \phi(1 - \phi)$
Tangens hiperboliczny	$\phi(x) = \tanh(x)$	$\phi' = 1 - \phi^2$
Eksponencjalna	$\phi(x) = \exp(x)$	$\phi' = \phi$
Skokowa	$\phi(x) = \mathbf{1}_{\{x \geq 0\}}$	
ReLU	$\phi(x) = x \mathbf{1}_{\{x \geq 0\}}$	

Pierwsze trzy funkcje z tabeli to gładkie funkcje z prostymi pochodnymi. Proste pochodne funkcji są dużą zaletą w algorytmie gradientu prostego, ponieważ nie komplikują złożoności obliczeniowej algorytmu. Poniżej przedstawiamy kilka uwag na temat funkcji aktywacji z tabeli 2.1.

- Funkcja logistyczna: Może być uzyskana z tangensa hiperbolicznego przez $\phi(x) = (\tanh(x/2) + 1)/2$. Również jest S-kształtna, ale przyjmuje wartości $(0; 1)$. Jest funkcją odwrotną do funkcji związku w modelu regresji logistycznej.
- Funkcja tangensa hiperbolicznego: Jest to S-kształtna funkcja aktywacji, ciągła i różniczkowalna. Jej przeciwdziedzina to $(-1; 1)$.
- Funkcja skokowa: W zasadzie nie jest wykorzystywana w rzeczywistych zastosowaniach. Natomiast jest często podawana jako przykład w dydaktyce ze względu na łatwość w interpretacji.

- Funkcja ReLU: Jest to najczęściej wybierana funkcja w społeczności zajmującej się uczeniem głębokim. Funkcja ta różni się od pozostałych brakiem pochodnej w punkcie 0, jednakże w praktyce działa bardzo dobrze. W celu uniknięcia problemów z punktem 0 przyjmuje się $\phi(x) = x\mathbf{1}_{\{x \geq \varepsilon\}}$, dla wartości ε bliskich zeru.

Wróćmy do rozważań o modelu jednokierunkowej sztucznej sieci neuronowej i podsumujmy pewne informacje.

Definicja 2.1 (Warstwa sieci jednokierunkowej) Warstwa jednokierunkowej sztucznej sieci neuronowej z funkcją aktywacji ϕ to odwzorowanie $\mathbf{z}^{(m)} : \{1\} \times \mathbb{R}^{q_{m-1}} \rightarrow \{1\} \times \mathbb{R}^{q_m}$

$$\mathbf{z}^{(m)}(\mathbf{x}) = \left(1, z_1^{(m)}(\mathbf{x}), \dots, z_{q_m}^{(m)}(\mathbf{x})\right)^\top, \quad (2.6)$$

dla $1 \leq j \leq q_m$ neuronów określonych jako

$$z_j^{(m)}(\mathbf{x}) = \phi\left(\langle \mathbf{w}_j^{(m)}, \mathbf{x} \rangle\right) = \phi\left(\sum_{l=0}^{q_{m-1}} w_{l,j}^{(m)} x_l\right), \quad (2.7)$$

dla danych wag (połączeń) $\mathbf{w}_j^{(m)} = (w_{l,j}^{(m)})_{0 \leq l \leq q_{m-1}} \in \mathbb{R}^{q_{m-1}+1}$.

Każdy neuron $z_j^{(m)}(\mathbf{x})$ określa funkcję uogólnionego modelu liniowego z funkcją związku ϕ^{-1} i parametrem $\mathbf{w}_j^{(m)} \in \mathbb{R}^{q_{m-1}+1}$ dla cech $\mathbf{x} \in \{1\} \times \mathbb{R}^{q_{m-1}}$. Te funkcje uogólnionego modelu liniowego mogą być interpretowane jako kompresja danych, to znaczy, że w każdym neuronie q_{m-1} -wymiarowa cecha \mathbf{x} jest rzutowana na liczbę rzeczywistą $\langle \mathbf{w}_j^{(m)}, \mathbf{x} \rangle \in \mathbb{R}$, która następnie jest nieliniowo aktywowana przez ϕ . Ponieważ prowadzi to do znacznej utraty informacji, wykonujemy tę procedurę kompresji danych q_m razy w warstwie $\mathbf{z}^{(m)}$, tak aby każdy neuron w $(z_j^{(m)}(\mathbf{x}))_{1 \leq j \leq q_m}$ reprezentował inną projekcję wejścia \mathbf{x} . Wybierając odpowiednie wagi $\mathbf{w}_j^{(m)}$ będziemy mogli wydobyć kluczowe informacje o cechach \mathbf{x} .

Definicja 2.2 (Jednokierunkowa sztuczna sieć neuronowa) Jednokierunkowa sztuczna sieć neuronowa o głębokości $d \in \mathbb{N}$ to sieć neuronowa skonstruowana ze złożenia d jednokierunkowych warstw $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(d)}$ uzyskując odwzorowanie $\mathbf{z}^{(d:1)} : \{1\} \times \mathbb{R}^{q_0=q} \rightarrow \{1\} \times \mathbb{R}^{q_d}$

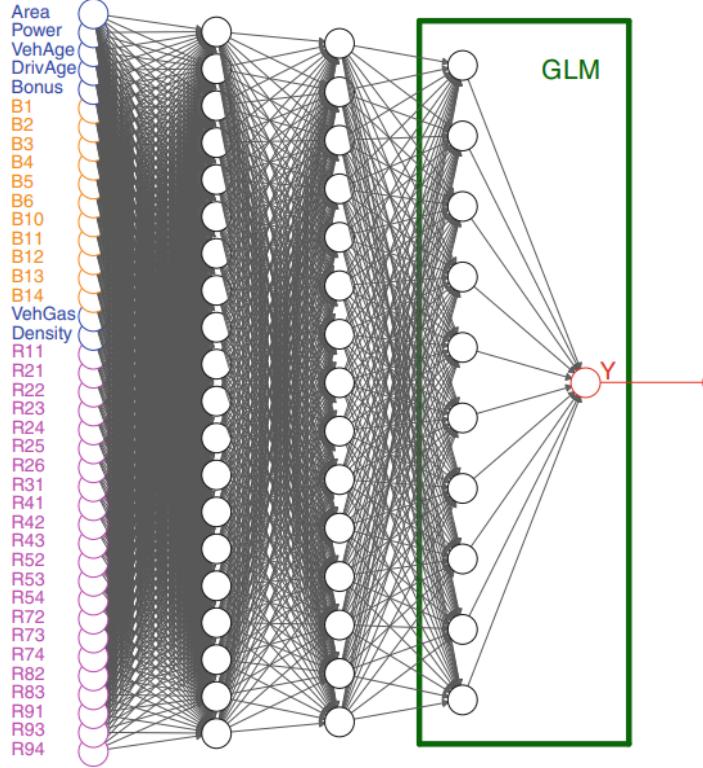
$$\mathbf{z}^{(d:1)}(\mathbf{x}) = (\mathbf{z}^{(d)} \circ \dots \circ \mathbf{z}^{(1)})(\mathbf{x}). \quad (2.8)$$

Wybierając ściśle monotoniczną i gładką funkcję związku g i parametr $\beta \in \mathbb{R}^{q_d+1}$ uzyskujemy funkcję modelu jednokierunkowej sztucznej sieci neuronowej

$$\mu(\mathbf{x}) = g^{-1}\left(\langle \beta, \mathbf{z}^{(d:1)}(\mathbf{x}) \rangle\right). \quad (2.9)$$

Taka funkcja modelu jednokierunkowej sztucznej sieci neuronowej ma parametr, nazywany parametrem sieci, składający się ze wszystkich wag oraz parametrów uogólnionego modelu liniowego, tzn. $\vartheta = (\mathbf{w}_1^{(1)}, \dots, \mathbf{w}_{q_d}^{(d)}, \beta) \in \mathbb{R}^r$ o wymiarze

$$r = \sum_{m=1}^d q_m(q_{m-1} + 1) + (q_d + 1). \quad (2.10)$$



Rysunek 2.5: Schemat przykładowej jednokierunkowej sztucznej sieci neuronowej o głębokości $d = 3$ z wymiarami $(q_1, q_2, q_3) = (20, 15, 10)$ i wymiarze danych wejściowych $q_0 = 40$. Daje nam to parametr sieci $\vartheta \in \mathbb{R}^r$ wymiaru $r = 1306$. Z uwagi na czytelność, na rysunku nie umieszczono wag połączeń oraz parametrów uogólnionego modelu liniowego. Źródło: [9].

Na rys. 2.5 przedstawiono schemat przykładowej jednokierunkowej sztucznej sieci neuronowej (czyli wielowarstwowego perceptronu) o głębokości $d = 3$ z wymiarami $(q_1, q_2, q_3) = (20, 15, 10)$ i wymiarze danych wejściowych $q_0 = 40$. Wówczas parametr sieci $\vartheta \in \mathbb{R}^r$ ma wymiar $r = 1306$. Po lewej stronie rysunku mamy cechy wejściowe $\mathbf{x} \in \mathcal{X} \subset \{1\} \times \mathbb{R}^{q_0}$, które następnie są przetwarzane przez trzy warstwy jednokierunkowej sztucznej sieci neuronowej, gdzie czarne okręgi symbolizują neurony $z_k^{(m)}$. Trzecia warstwa sieci $\mathbf{z}^{(3)}$ ma wymiar $q_3 = 10$, dzięki której uzyskujemy wyuczone reprezentacje $\mathbf{z}^{(3:1)}(\mathbf{x}) \in \{1\} \times \mathbb{R}^{q_3}$ cech \mathbf{x} . Następnie reprezentacje wykorzystywane są w kroku uogólnionego modelu liniowego oznaczonego zielonym prostokątem. Możemy wykonywać zarówno zadanie regresji, jak i klasyfikacji (także wieloklasowej), jest to zdeterminowane przez wybór odpowiedniej funkcji związku g .

2.6 Algorytm gradientu prostego ze wsteczną propagacją błędu

W niniejszym podrozdziale prezentujemy najpopularniejsze techniki treningu sztucznych sieci neuronowych. Nie omawiamy tu wszystkich możliwych algorytmów z uwagi na ich dużą liczbę. Przedstawiamy natomiast ogólny schemat postępowania (algorytm gradientu prostego), który jest podstawą bardziej zaawansowanych metod.

Algorytm gradientu prostego z wykorzystaniem wstecznej propagacji błędu odbywa się w siedmiu podstawowych krokach, które przedstawiamy poniżej.

1. Losowa inicjalizacja wag sieci

Na etapie początkowym treningu inicjalizowane są wagi w sieci neuronowej. Proces inicjalizacji może przyjąć różne formy, jednakże najczęściej stosuje się losowe nadawanie wartości wagom połączeń. Losowe inicjalizacje mają na celu wprowadzenie pewnego stopnia przypadkowości do modelu, co jest kluczowe w celu uniknięcia wpadania w lokalne minimum funkcji kosztu.

Istnieją różne techniki inicjalizacji, które są dostępne w celu zoptymalizowania inicjalizacji wag. Jednym z popularnych podejść jest inicjalizacja wag poprzez losowanie wartości z rozkładu normalnego o określonych parametrach.

2. Przebieg w przód

Po zainicjowaniu wag dane wejściowe przechodzą przez sieć neuronową, podążając od warstwy wejściowej w kierunku warstwy wyjściowej. Ten proces, nazywany przebiegiem w przód, obejmuje kilka istotnych etapów.

Po pierwsze dane wejściowe są przekazywane przez kolejne warstwy sieci. Każda warstwa przyjmuje dane od poprzedniej warstwy i przetwarza je, wykonując określone przekształcenia. W skrócie: w każdej warstwie sieci wykonuje się działanie iloczynu skalarnego, który jako argumenty przyjmuje dane wejściowe oraz wagi połączeń warstw, które zostały wcześniej zainicjowane.

Następnie po obliczeniu tej sumy ważonej, wyniki są poddawane funkcjom aktywacji. Funkcje te wprowadzają nieliniowość do modelu, co pozwala sieci na wychwycenie bardziej złożonych zależności i ekstrakcję cech z danych. Przykładowe funkcje aktywacji przedstawiliśmy w tabeli 2.1. Formalny opis tego procesu szczegółowo przedstawiliśmy w podrozdziale 2.5 (zob. def. 2.1).

Na koniec uzyskujemy wynik z warstwy wyjściowej, która stanowi predykcję modelu. W tej warstwie dokonuje się ostatecznych decyzji na podstawie przetworzonych danych wejściowych. W przypadku zadania klasyfikacji, może to oznaczać przypisanie danych wejściowych do określonych klas.

3. Obliczanie funkcji kosztu

Po uzyskaniu prognoz od sieci neuronowej, przeprowadzany jest proces oceny, który ma na celu zrozumienie, jak dobrze model radzi sobie z przewidywaniem rzeczywistych etykiet danych treningowych. Proces ten obejmuje kilka kroków.

Na początek następuje porównywanie prognoz z rzeczywistością. Prognozy uzyskane od modelu są porównywane z rzeczywistymi etykietami danych treningowych, takimi jak klasy w przypadku zadania klasyfikacji lub konkretne wartości w przypadku zadania regresji. Do obu tych wartości stosowana jest funkcja kosztu, która daje nam miarę niedopasowania pomiędzy wartościami prognozowanymi a wartościami rzeczywistymi. Ta miara określa, jak bardzo model różni się od rzeczywistości. W ogólności im niższa wartość funkcji kosztu, tym lepsze są prognozy.

Funkcja kosztu jest zadana z góry dla danego procesu treningu. Istnieje wiele rodzajów funkcji kosztu i różnią się one w zależności od zadania klasyfikacji i regresji. Przykładowe funkcje kosztu dla zadania klasyfikacji, które stosujemy w części eksperymentalnej pracy (zob. podrozdz. 4.6), zdefiniowaliśmy w podrozdziale 1.1. Są to: binarna entropia krzyżowa (zob. def. 1.8) oraz kategoryalna entropia krzyżowa (zob. def. 1.9). Wartość funkcji kosztu oblicza się na podstawie pewnej liczby przetworzonych przykładów danych. Przykłady określamy jako partię (ang. *batch*). Funkcja kosztu jako argumenty przyjmuje wyniki sieci neuronowej (prognozy) dla każdego przykładu z partii oraz rzeczywiste wartości etykiet danych każdego przykładu z partii i zwraca wartość skalarną. Ma zatem charakter funkcjonału.

Celem treningu jest minimalizacja wartości funkcji kosztu. W praktyce oznacza to dostosowywanie wag w sieci neuronowej w taki sposób, aby zmniejszać niedopasowanie modelu do rzeczywistych danych i tym samym polepszać jakość prognoz.

4. Wsteczna propagacja błędu

Wsteczna propagacja błędu, znana również jako przebieg wstecz, stanowi kluczowy etap treningu sieci neuronowej. Po obliczeniu funkcji kosztu, algorytm spadku gradientu rozpoczyna proces propagacji wstecznej, który ma na celu dokładne dostosowanie wag w sieci, czyli parametru sieci ϑ .

Na początek algorytm oblicza gradienty funkcji kosztu względem wszystkich wag w sieci. Gradienty te mówią nam, jak bardzo i w jakim kierunku funkcja kosztu zmienia się w odpowiedzi na zmiany w wagach. Wartości te są bardzo istotne dla określenia, jak dostosować wagi w celu minimalizacji funkcji kosztu.

Proces propagacji wstecznej jest iteracyjny i obejmuje przechodzenie wstecz przez wszystkie warstwy sieci, zaczynając od warstwy wyjściowej i kończąc na warstwie wejściowej. Dla każdej warstwy obliczane są gradienty na podstawie gradientów z warstwy poprzedniej.

Optymalna wartość parametru sieci ϑ jest znajdywana za pomocą minimalizacji funkcji kosztu J_{ϑ} . Niestety problem ten nie może być rozwiązany analitycznie. Z tego powodu poszukujemy parametr sieci $\hat{\vartheta}$, który zapewnia „niewielką” wartość J_{ϑ} .

Algorytm stara się krok po kroku lokalnie poprawiać pozycję, zmieniając wagi w parametrze sieci w kierunku maksymalnego lokalnego spadku funkcji kosztu.

Wzór na obliczenie gradientu $\nabla_{\vartheta} J_{\vartheta(t)}$ dla danego parametru sieci, który składa się z wag sieci, wynika z zastosowanej funkcji kosztu i funkcji aktywacji w danej warstwie. Algorytm wykorzystuje regułę łańcuchową do efektywnego przekazywania gradientów wstecz przez sieć.

Kalkulacja gradientu funkcji kosztu sprowadza się do wyznaczenia gradientu

$$\nabla_{\boldsymbol{\vartheta}} \left(\langle \boldsymbol{\beta}, \mathbf{z}^{(d:1)}(\mathbf{x}) \rangle \right) = \nabla_{\boldsymbol{\vartheta}} \left(\langle \boldsymbol{\beta}, (\mathbf{z}^{(d)} \circ \dots \circ \mathbf{z}^{(1)}) (\mathbf{x}) \rangle \right), \quad (2.11)$$

względem parametru sieci $\boldsymbol{\vartheta} = (\mathbf{w}_1^{(1)}, \dots, \mathbf{w}_{q_d}^{(d)}, \boldsymbol{\beta})^\top \in \mathbb{R}^r$ i gdzie każda warstwa sieci \mathbf{z}^m zawiera wagi $\mathcal{W} = (\mathbf{w}_1^{(m)}, \dots, \mathbf{w}_{q_m}^{(m)}) \in \mathbb{R}^{(q_{m-1}+1) \times q_m}$. To właśnie do (2.11) stosowana jest reguła łańcuchowa.

Wykorzystanie reguły łańcuchowej do (2.11) pozwala na wyznaczenie

$$\text{diag} \left(\phi' \left(\langle \mathbf{w}_{j_m}^{(m)}, \mathbf{z}^{(m-1:1)}(\mathbf{x}) \rangle \right) \right)_{1 \leq j_m \leq q_d}, \quad (2.12)$$

gdzie ϕ' to pochodna funkcji aktywacji. Widzimy zatem dlaczego tak ważnym założeniem jest różniczkowalność funkcji aktywacji, o czym wspominaliśmy w podrozdziałach 2.3 i 2.5. Wartość (2.12) jest bezpośrednio wykorzystywana w propagacji wstępnej i finalnie aktualizacji wag. Szczegółowy opis algorytmu wstępnej propagacji, na przykładzie funkcji aktywacji tangensa hiperbolicznego, można znaleźć w [31, podrozdz. 7.2].

5. Aktualizacja wag

Proces aktualizacji wag w sieci neuronowej jest niezwykle istotny w treningu modeli. Po obliczeniu gradientów w trakcie propagacji wstępnej, algorytm spadku gradientu przystępuje do dostosowania wszystkich wag w modelu.

W ramach aktualizacji, każda waga w sieci jest poddawana modyfikacji. Obejmuje to wagi dla neuronów każdej warstwy, od warstwy wyjściowej aż po warstwę wejściową. Dlatego też proces aktualizacji ma wpływ na wszystkie parametry modelu.

Ogólny wzór aktualizacji wag opiera się na różnicy między aktualną wagą a iloczynem współczynnika uczenia i gradientu funkcji kosztu względem tej wagi. Matematycznie jest to wyrażone jako:

$$\boldsymbol{\vartheta}^{(t+1)} = \boldsymbol{\vartheta}^{(t)} - \varrho \nabla_{\boldsymbol{\vartheta}} J_{\boldsymbol{\vartheta}^{(t)}}(\mathbf{Y}_{\text{Prognozowane}}, \mathbf{Y}_{\text{Rzeczywiste}}), \quad (2.13)$$

gdzie $\boldsymbol{\vartheta}^{(t+1)}$ to zaktualizowany parametr sieci, $\boldsymbol{\vartheta}^{(t)}$ to stary parametr sieci, ϱ to współczynnik uczenia, a $\nabla_{\boldsymbol{\vartheta}} J(\mathbf{Y}, \boldsymbol{\vartheta}^{(t)})$ to gradient funkcji kosztu J uzyskany za pomocą propagacji wstępnej, zależny od danych prognozowanych $\mathbf{Y}_{\text{Prognozowane}}$, rzeczywistych $\mathbf{Y}_{\text{Rzeczywiste}}$ i parametru sieci $\boldsymbol{\vartheta}^{(t)}$.

Wagi są uaktualniane w kierunku przeciwnym do gradientu, co ma na celu minimalizację funkcji kosztu. Wartość współczynnika uczenia (ang. *learning rate*) kontroluje, jak duże będą kroki aktualizacji wag.

Współczynnik uczenia ma zasadnicze znaczenie w procesie aktualizacji wag. Jest to parametr, który kontroluje, jak duże kroki zostaną podjęte podczas aktualizacji wag. Wartość współczynnika uczenia musi być dobrze dostosowana, ponieważ ma to wpływ na zbieżność treningu. Wartość zbyt duża może powodować oscylacje lub utrudnić osiągnięcie minimum funkcji kosztu, podczas gdy wartość zbyt mała może spowolnić proces uczenia i utknąć w lokalnym minimum.

Aktualizacja wag pozwala modelowi dostosowywać swoje parametry w odpowiedzi na błędy predykcji.

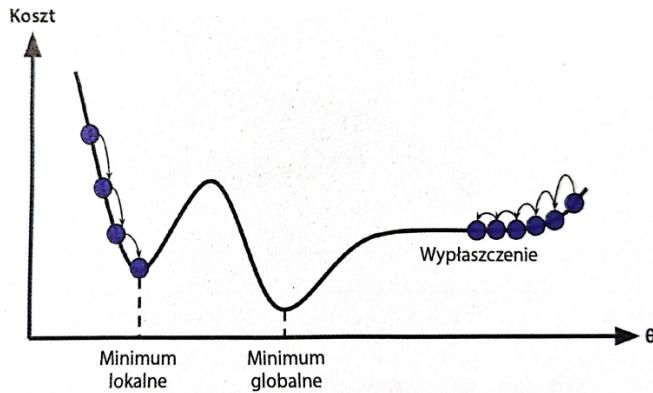
6. Powtarzanie iteracji

Kroki 2-5, które obejmują obliczanie gradientów, aktualizację wag i ocenę modelu, odbywają się w każdej iteracji treningu, czyli tzw. epok (ang. *epoch*). Jedna epoka stanowi pełne przejście przez cały zestaw danych treningowych dostępny dla modelu. Minimalizujemy funkcji kosztu, co oznacza, że wagi są systematycznie dostosowywane, aby model coraz lepiej przewidywał rzeczywiste etykiety danych treningowych.

Celem tego procesu treningu jest minimalizacja wartości funkcji kosztu. Jest to cel iteracyjny, co oznacza, że dążymy do stopniowego zmniejszania błędu modelu w miarę postępu w treningu. To iteracyjne podejście pozwala modelowi na dostosowanie wag w trakcie treningu.

W trakcie każdej epoki model jest wielokrotnie uczyony na podstawie danych treningowych. Ten cykl powtarza się, aż model osiągnie zadowalającą wydajność i minimalizację funkcji kosztu. Dlatego też trening sieci neuronowej jest procesem ewolucyjnym, w którym model stopniowo dostosowuje swoje parametry, aby stawać się coraz lepiej dopasowanym do konkretnego zadania.

Zbieżność algorytmu spadku gradientu zależy w dużej mierze od kształtu funkcji kosztu i wartości współczynnika uczenia. Na rys. 2.6 przedstawiamy potencjalne problemy związane z algorymem gradientu prostego. Możliwe jest, że algorytm „utknie” w minimum lokalnym, nie osiągając minimum globalnego. Widzimy to na rysunku po lewej stronie. Inną możliwością jest,



Rysunek 2.6: Potencjalne problemy w algorytmie gradientu prostego. Źródło: [7].

że algorytm może wymagać dużo czasu, zanim osiągnie minimum funkcji kosztu, albo w ogóle nie dojdzie do tego minimum, jeśli zakończy działanie zbyt szybko. Te sytuacje widzimy po prawej stronie rysunku. Jednakże istnieje wiele funkcji kosztu, które są wypukłe oraz spełniają warunek Lipschitza, co oznacza, że nie istnieją lokalne minima tych funkcji, tylko jedno minimum globalne np. funkcja błędu średniokwadratowego dla regresji liniowej.

7. Zakończenie treningu

Trening modelu może zostać zakończony w zależności od przyjętej strategii, na przykład po określonej liczbie epok lub w odpowiedzi na konkretne warunki.

Przykładowe kryterium zakończenia treningu to osiągnięcie zadanej wydajności na danych walidacyjnych. To oznacza, że jego zdolność do dokładnego przewidywania na nowych danych jest wystarczająco wysoka.

Innym kryterium zakończenia treningu może być brak znaczącej poprawy funkcji kosztu przez kilka kolejnych epok. Gdy model przestaje istotnie polepszać swoje predykcje, może to sugerować, że doszedł do punktu, w którym dalszy trening jest mało efektywny.

Jedną z technik zakończenia treningu, która polega na monitorowaniu metryk treningowych i walidacyjnych i zatrzymaniu treningu, gdy wykryte zostaną sygnały przeuczenia lub braku postępu w wydajności modelu, jest metoda wczesnego zatrzymania (ang. *early stopping*). Szczegółowy opis techniki wczesnego zatrzymania można znaleźć w [9, rozdz. 7].

Po zakończeniu procesu treningu, wytrenowany model staje się użytecznym narzędziem, które może być wykorzystywane do dokonywania predykcji na nowych danych. Model ten posiada wagę i parametry, które zostały dostosowane w procesie uczenia, co umożliwia mu przewidywanie na ich podstawie wartości szukanej dla nowych danych wejściowych. Ostatecznie celem treningu jest stworzenie modelu zdolnego do dokładnych i skutecznych predykcji w kontekście konkretnej dziedziny lub zadania.

Stochastyczny gradient prosty

Istnieje wiele modyfikacji algorytmu gradientu prostego. Jedną z najpopularniejszych jest algorytm stochastycznego gradientu prostego, który stosujemy w części eksperymentalnej pracy (zob. podrozdz. 4.6).

W algorytmie stochastycznego gradientu prostego, podczas każdej iteracji treningu, dokonujemy losowego wyboru pojedynczego przykładu lub małego mini-partii (ang. *mini batch*) przykładów ze zbioru treningowego. Następnie obliczamy gradient funkcji kosztu na podstawie tego wylosowanego przykładu lub mini-partii i aktualizujemy wagę modelu. Taka losowa procedura wyboru przykładów czyni algorytm stochastycznego gradientu prostego bardziej efektywnym obliczeniowo w porównaniu do tradycyjnego algorytmu spadku gradientu. Niemniej jednak, charakter losowy może wprowadzać większą niestabilność, gdyż pojedyncze przykłady mogą znacząco zmieniać kierunek aktualizacji wag. Pomimo tych potencjalnych niestabilności, podejście to jest szczególnie korzystne w przypadku dużych zbiorów danych, ponieważ przyspiesza proces dążenia wag modelu do optymalnych wartości.

Rozdział 3

Charakterystyka uczenia głębokiego w przetwarzaniu obrazu

W niniejszym rozdziale przedstawiona została teoria stanowiąca podstawę przetwarzania obrazu w kontekście sztucznych sieci neuronowych. Omówiono istotne koncepcje, które znajdują zastosowanie w sieciach splotowych, wraz z przedstawieniem ogólnej struktury tych sieci. W podrozdziałach 3.3-3.6 dokonano intuicyjnego wyjaśnienia poszczególnych zagadnień dotyczących sieci splotowych, natomiast w podrozdziale 3.7 dokonano sformalizowania omawianych idei. Przykład architektury sieci splotowej, wraz z jej formalnym opisem i implementacją w języku Python, został zaprezentowany w podrozdziale 3.8. Treść tego rozdziału opiera się na źródłach literaturowych [1, 2, 7, 9, 14, 31, 32].

3.1 Wprowadzenie do zadania rozpoznawania obrazów

Rozpoznawanie obrazów to jeden z kluczowych obszarów w dziedzinie uczenia maszynowego oraz sztucznej inteligencji. Jest to zadanie, które polega na tworzeniu zaawansowanych algorytmów i modeli umożliwiających komputerom analizowanie, interpretowanie i rozumienie zawartości obrazów z intencją osiągnięcia częścią cech ludzkiego widzenia. Stanowi ono nieodzowną część wielu dziedzin, takich jak medycyna, przemysł, rozrywka lub bezpieczeństwo.

Zagadnienia rozpoznawania obrazów niesie ze sobą szereg wyzwań, wynikających z konieczności przetwarzania ogromnej ilości informacji zawartych w obrazach. Obrazy są z natury złożone i często niejednoznaczne, co wymaga zastosowania zaawansowanych technik i narzędzi, aby dokładnie analizować ich treść. Głównym celem rozpoznawania obrazów jest rozwinięcie komputerowej zdolności do identyfikowania obiektów, wzorców, atrybutów oraz relacji obecnych na obrazach. To umożliwia komputerom podejmowanie informowanych decyzji na podstawie wizualnych danych.

W ostatnich latach dokonano znaczącego postępu w dziedzinie rozpoznawania obrazów, głównie dzięki zaawansowanym algorytmom i modelom opartym na technikach uczenia maszynowego, zwłaszcza uczeniu głębokiemu. Modele takie jak splotowe sieci neuronowe stały się nieodzownym narzędziem do automatycznego wyodrębniania cech z obrazów, rozpoznawania obiektów oraz ich klasyfikacji.

W niniejszym rozdziale skupimy się na wprowadzeniu do zagadnienia rozpoznawania obrazów w kontekście głębokiego uczenia maszynowego, czyli sieci splotowych. Rozważmy fundamentalne koncepcje, napotykane wyzwania oraz zastosowane techniki w tym obszarze. Omówimy proces ekstrakcji cech oraz reprezentacji obrazów, przedstawimy strukturę i działanie modeli sieci splotowych.

3.2 Ogólna architektura sieci splotowej

W tym podrozdziale przedstawiamy schemat ogólnej architektury sieci splotowej. Niniejszy podrozdział powstał na podstawie [7, 9].

Typowe konstrukcje architektur sieci splotowych zwykle składają się z kilku warstw splotowych, następnie warstwy łączącej, a po niej ponownie z kilku warstw splotowych, kontynuując ten cykl. Na rys. 3.1 przedstawiono schemat przykładowej architektury sieci splotowej. Rozmiar wejściowego obrazu stopniowo maleje w miarę przechodzenia przez kolejne warstwy sieci, ale jednocześnie zwiększa się liczba reprezentacji obrazu przetworzonego przez warstwy sieci.

Po ostatniej warstwie łączącej następuje spłaszczenie reprezentacji danych wejściowych, które są przekazywane do jednokierunkowej sztucznej sieci neuronowej, którą w kontekście architektury

sieci splotowej nazywamy czasami warstwą w pełni połączoną. W rezultacie uzyskujemy realizację zadania uczenia maszynowego – zwykłe zadania klasyfikacji. Kolejne podrozdziały przedstawiają poszczególne komponenty architektury sieci splotowej.

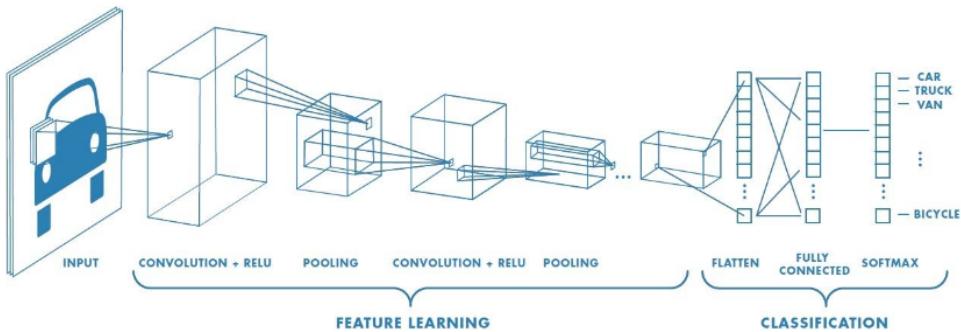
Należy podkreślić, że istnieją różne sposoby patrzenia na sieć splotową w zależności od jej warstw. Pierwszy z powszechnie stosowanych sposobów to spojrzenie na warstwę splotową jako na całość składającą się z kolejnych etapów. Kolejne etapy w tej interpretacji to: etap splotu (wykonanie operacji splotu na danych wejściowych w celu uzyskania mapy cech), etap detekcji (nadawanie przekształceniu nielinowości za pomocą funkcji aktywacji zastosowanych do wartości wyjściowych z etapu splotu) oraz następująca po tych etapach warstwa łącząca.

Inny sposób patrzenia na sieć splotową to spojrzenie na każdy etap, jako osobną warstwę. W ten sposób mamy najpierw warstwę wejściową, dalej warstwę splotową, która wykonuje operację splotu na danych wejściowych. Kolejna warstwa to warstwa przekształcająca wynik z warstwy splotowej przy zastosowaniu funkcji aktywacji. Ostatnia warstwa to warstwa łącząca.

W niniejszej pracy stosujemy pierwsze podejście do architektury sieci splotowej. W ten sposób najprostsza architektura sieci splotowej ma następującą postać.

1. Warstwa wejściowa: Jest to etap wczytania danych, najczęściej będzie to kolorowy obraz (na rys. 3.1 podpisana jako *INPUT*)).
2. Warstwa splotowa: Wykonywana jest tu operacja splotu, wraz z następującymi po sobie kolejnymi operacjami (na rys. 3.1 podpisana jako *CONVOLUTION + RELU*)).
3. Warstwa łącząca: Zastosowana do wyniku warstwy splotowej (na rys. 3.1 podpisana *POOLING*).
4. Warstwa spłaszczająca: Nadaje reprezentacjom uzyskanym przez poprzednie warstwy formę wektora, który służy jako wejście do warstwy w pełni połączonej (na rys. 3.1 podpisana jako *FLATTEN*)).
5. Warstwa w pełni połączona: jest to w rzeczywistości jednokierunkowa sieć neuronowa o dowolnej głębokości (na rys. 3.1 podpisana jako *FULLY CONNECTED*)).
6. Warstwa wyjściowa: Następuje tu wykonanie predykcji zgodnie z przyjętym zadaniem dla sieci (na rys. 3.1 podpisana jako *SOFTMAX*)).

Na rys. 3.1 mamy tu do czynienia z klasyfikacją obrazu pojazdu, jest to obraz o 3 kanałami (np. odpowiadającym kolorom: czerwonemu, zielonemu, niebieskiemu). Warstwy podane są zgodnie z konwencją przedstawioną powyżej. Kolejne warstwy splotowe i łączące naprzemiennie razem służą do wykrywania cech obrazu wejściowego (*FEATURE LEARNING*).



Rysunek 3.1: Schemat architektury sieci splotowej. Źródło: [24].

3.3 Operator splotu w przetwarzaniu obrazu

W tym podrozdziale wyjaśniamy działanie operatoru splotu wykorzystywanego w sieciach splotowych. Niniejszy podrozdział powstał w oparciu o [1, 2, 9].

Splotowe sieci neuronowe swoją nazwę zawdzięczają operacji matematycznej zwanej splotem lub konwolucją (ang. *convolution*). Zrozumienie mechanizmu działania operatora splotu jest kluczowe w zrozumieniu mechanizmu działania splotowej sieci neuronowej, z uwagi na ten fakt w tej sekcji wyjaśnimy działanie splotu wykorzystywanego w sieciach splotowych. Na ogół splot w sieci

splotowej nie odpowiada dokładnie operacji splotu stosowanej w matematyce czystej czy inżynierii, lecz jest modyfikacją jej dyskretniej wersji.

Ogólna wersja splotu opiera się na definicji wykorzystującej funkcje z przestrzeni funkcji całkowalnych w sensie Lebesgue'a,

Definicja 3.1 (Splot) Niech $f: \mathbb{R} \rightarrow \mathbb{R}$ i $g: \mathbb{R} \rightarrow \mathbb{R}$ będą funkcjami z przestrzeni $L^1(\mathbb{R})$, czyli są całkowalne w sensie Lebesgue'a. Wówczas *splot* tych funkcji wyraża się wzorem:

$$(f * g)(t) = \int_{\mathbb{R}} f(t - \tau)g(\tau)d\tau. \quad (3.1)$$

Tak zdefiniowany splot nie znajduje jednak prostego zastosowania w zadaniach wymagających pracy na danych i operacji komputerowych. Bardziej użyteczna wersja konwolucji ma charakter dyskretny ze względu na to, że w ogólności dane w komputerze są zdyskretyzowane. Przejdzmy do definicji.

Definicja 3.2 (Splot dyskretny) Niech $t \in \mathbb{Z}$ oraz niech $x: \mathbb{Z} \rightarrow \mathbb{R}$ oraz $w: \mathbb{Z} \rightarrow \mathbb{R}$. Wtedy *splot dyskretny* funkcji x oraz w to funkcja $x * w: \mathbb{Z} \rightarrow \mathbb{R}$ dana wzorem:

$$(x * w)(t) = \sum_{a \in \mathbb{Z}} x(a)w(t - a). \quad (3.2)$$

x i w są sygnałami wejściowymi, z punktu widzenia programistycznego będą to macierze, wektory, czy innego rodzaju kontenery o wartościach rzeczywistych.

W terminologii sieci splotowych pierwszy argument (tu x) określany jest jako wejście, a drugi argument (tu w) jako jądro (ang. *kernel*), jądro splotowe (ang. *convolutional kernel*) lub filtr (ang. *filter*), natomiast wynik operacji splotu powyższych określany jest mianem mapy cech (ang. *feature map*, zob. podrozdz. 3.5). W praktyce sieci splotowe jako wejście x wykorzystują wielowymiarowe tablice danych, a filtry w są wielowymiarowymi tablicami parametrów (wag). W dalszej części pracy te wielowymiarowe tablice określać będziemy tensorami. Każdy element funkcji x i w musi być osobno przechowywany w pamięci komputera, więc zakładać będziemy, że x i w mają wartości zero wszędzie poza skońzoną liczbą elementów, których wartości są zapisywane. Oznacza to, że nieskończoną sumę z (3.2) możemy zaimplementować jako skońzoną sumę elementów tensorów x i w . Operacji splotu możemy wykonywać po dwóch (lub więcej) osiach jednocześnie.

Definicja 3.3 (Splot w sieci splotowej) Założymy, że nasze dane wejściowe to dwuwymiarowy obraz oraz $I: \mathbb{Z}^2 \rightarrow \mathbb{R}$. Filtr określamy jako dwuwymiarową tablicę o wymiarze $a \times b$ oraz stwarzającą z nim funkcję $K: \mathbb{Z}^2 \rightarrow \mathbb{R}$. Wtedy wzór na *splot w sieci splotowej* przedstawia się następująco

$$(I * K)(i, j) = \sum_{m=1}^a \sum_{n=1}^b I(m, n)K(i - m, j - n). \quad (3.3)$$

W powyższej definicji I podaje nam wartość (np. natężenie koloru) dla danych współrzędnych piksela obrazu, a K podaje wartość filtra dla określonych współrzędnych.

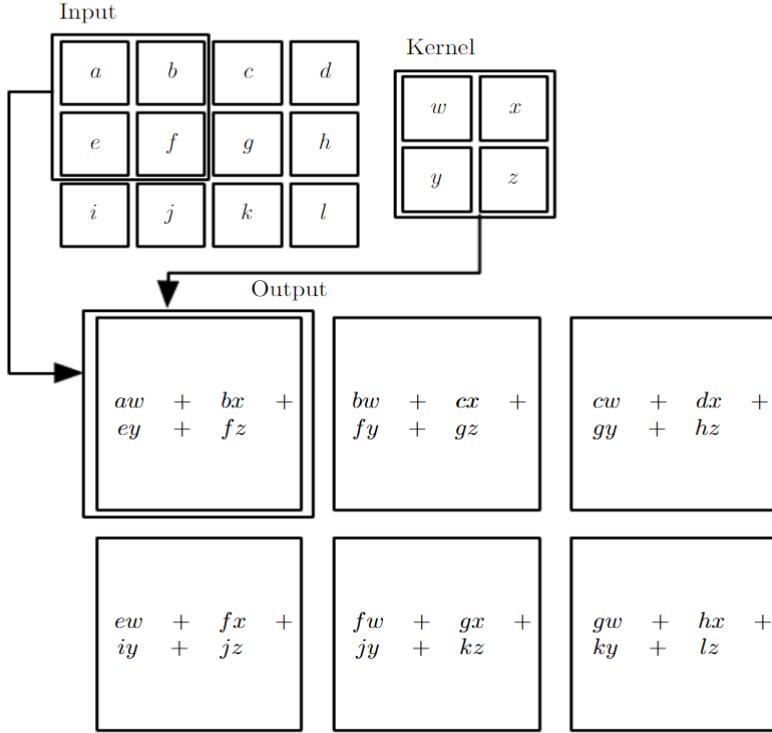
Warto zaznaczyć, że splot jest operacją przemienną, to znaczy: $(I * K)(i, j) = (K * I)(i, j)$. W praktyce często wykonywana jest implementacja operacji $(K * I)(i, j)$ zamiast $(I * K)(i, j)$.

W przypadku $(K * I)(i, j)$ „odwracamy” filtr względem wejścia, w tym sensie, że zwiększaając wartość m , zwiększymy indeks wejścia, ale zmniejszamy indeks filtru. Odwracanie filtru jest konieczne, aby uzyskać własność przemienności splotu. Chociaż własność ta jest użyteczna przy przeprowadzaniu dowodów rozumowań, to zazwyczaj nie jest ona ważną cechą implementacji sieci neuronowych. Zamiast tego wiele bibliotek sieci neuronowych implementuje podobną funkcję nazywaną korelacją krzyżową (ang. *cross-correlation*), która jest tożsama z splotem, ale bez odwracania filtru.

Definicja 3.4 (Korelacja krzyżowa) Przy założeniach z definicji 3.3 *korelacja krzyżowa* wyraża się wzorem

$$(I * K)(i, j) = \sum_{m=1}^a \sum_{n=1}^b I(i + m, j + n)K(m, n) \quad (3.4)$$

Według konwencji nazewnictwa uczenia maszynowego, korelacja krzyżowa nazywana jest splotem, w niniejszej pracy również stosować będziemy tę konwencję. W powyższej definicji widzimy, że „odwrócenie” filtru polega na zmianie w indeksowaniu dla funkcji I oraz K , a nie na samej zmianie kolejności w splocie funkcji. Na rys. 3.2 przedstawiono przykład działania splotu bez „odwracania” filtru.



Rysunek 3.2: Ilustracja działania splotu bez odwracania filtru. W powyższym przykładzie filtr ma wymiar 2×2 . Zakres i oraz j jest ograniczony do tych elementów, dla których filtr w konkretnym położeniu zawiera się w obrazie, czyli nie „wystaje” poza obraz, a więc dla elementów a, b, c, e, f, g z wejścia. Źródło: [9].

3.4 Warstwa splotowa

W niniejszym podrozdziale omawiamy koncepcję warstwy splotowej w sieci splotowej. Podrozdział powstał na podstawie [7, 9].

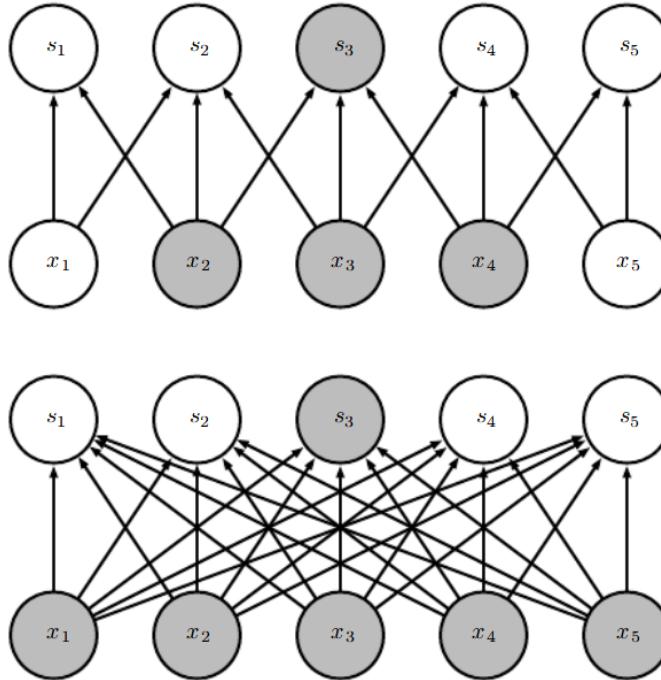
Warstwa splotowa (ang. *convolutional layer*), lub inaczej warstwa konwolucyjna, to najistotniejszy element sieci splotowej. Posiada ona szereg cech, które wyróżniają ją od tradycyjnych sieci takich jak np. perceptron. Omówimy kolejno trzy istotne charakterystyki tej warstwy: rozproszone interakcje (ang. *sparse interactions*), współdzielanie parametrów (ang. *parameter sharing*) oraz reprezentacje ekwiwariantne (ang. *equivariant representations*).

W tradycyjnych warstwach sieci neuronowych każda jednostka (neuron) wyjściowa jednej warstwy oddziałuje na każdą jednostkę wejściową następnej warstwy poprzez mnożenie macierzowe z macierzą parametrów. W przeciwieństwie do tego sieci splotowe zazwyczaj posiadają rozproszone interakcje, nazywane również rozproszonym połączeniem (ang. *sparse connectivity*) lub rozproszonymi wagami (ang. *sparse weights*). Aby osiągnąć tę cechę, używa się jądra splotu o mniejszych rozmiarach niż dane wejściowe. Na rys. 3.3 przedstawiono ilustrację połączeń rozproszych.

Przykładowo, w przypadku przetwarzania obrazów, gdzie obraz wejściowy może mieć tysiące lub miliony pikseli, możemy wykrywać małe, istotne cechy, takie jak krawędzie, za pomocą jąder zajmujących tylko kilkadziesiąt lub kilkaset pikseli. To z kolei oznacza, że potrzebujemy przechoływać mniej parametrów, co zarówno zmniejsza wymagania pamięciowe modelu, jak i poprawia jego efektywność statystyczną. Oznacza to także, że w celu wyznaczenia wyjścia wymagana jest mniejsza liczba operacji niż w ujęciu tradycyjnym.

Te ulepszenia mają zwykle znaczący wpływ na efektywność. Jeśli istnieje m wejść i n wyjść, to mnożenie macierzowe wymaga $m \times n$ parametrów, a stosowane w praktyce algorytmy mają złożoność obliczeniową $O(mn)$. Jeśli ograniczymy liczbę połączeń, które każde wyjście może mieć, do k , to podejście z rozproszonym połączeniem wymaga tylko $k \times n$ parametrów i ma złożoność obliczeniową $O(kn)$. W wielu praktycznych zastosowaniach możliwe jest osiągnięcie dobrej wydajności w zadaniach uczenia maszynowego, utrzymując k o kilka rzędów wielkości mniejsze niż m . Pozwala to sieci na efektywne opisywanie złożonych interakcji między wieloma zmiennymi.

W kontekście przetwarzania obrazu oznacza to, że neurony z pierwszej warstwy nie mają połączenia z każdym pikselem obrazu wejściowego, lecz wyłącznie z liczbą pikseli ograniczoną przez wymiar filtru splotu. Neurony wejściowe, które są brane pod uwagę przez jądro, określone



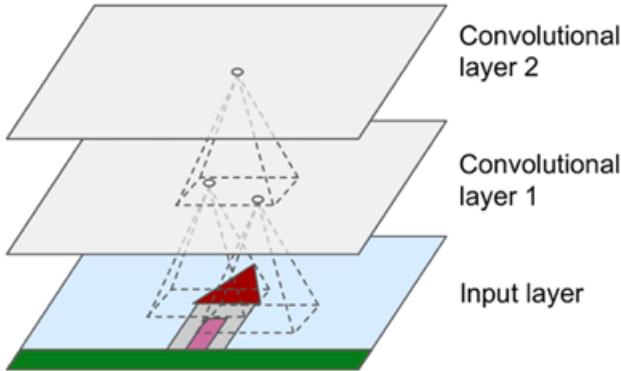
Rysunek 3.3: Ilustracja rozproszonych połączeń w warstwie splotowej. Kolorem szarym zaznaczono neuron wyjściowy s_3 i neurony wejściowe x , które wpływają na ten neuron. U góry rysunku: gdy s jest tworzony poprzez splot z filtrem o szerokości 3, tylko trzy wejścia wpływają na s_3 . U dołu rysunku: gdy s jest tworzony poprzez mnożenie macierzowe, połączenia nie są już rozproszone, dlatego wszystkie wejścia wpływają na s_3 , co może znacznie zwiększyć złożoność obliczeniową. Źródło: [9].

są mianem pola receptivejnego (ang. *receptive field*), oczywiście pole to ma taki sam rozmiar jak jądro splotu. Idąc dalej, każdy neuron w potencjalnej drugiej warstwie połączony jest z odpowiednim wycinkiem neuronów z pierwszej warstwy, tworząc kolejne pole receptivejne. Podkreślmy, że mówimy tu o polu receptivejnym pojedynczego neuronu z warstwy splotowej, gdyż suma mnożnościowa wszystkich neuronów ma połączenie do wszystkich pikseli obrazu wejściowego. Taka konstrukcja umożliwia koncentrowanie się na ogólnych cechach w pierwszej warstwie, aby następnie układać je w bardziej złożone kształty w kolejnych warstwach. Stanowi to o ogromnej sile sieci splotowych w przetwarzaniu obrazu. Na rys. 3.4 przedstawiono ilustrację warstw sieci splotowej z polami receptivejnymi.

Kolejną istotną cechą warstwy splotowej jest współdzielenie parametrów. Cechą ta polega na wykorzystywaniu tego samego parametru dla wielu funkcji w modelu. W tradycyjnej sieci neuronowej każdy element z macierzy wag jest wykorzystany tylko raz podczas obliczania wyniku warstwy. Jest on pomnożony przez jeden element wejścia i nie jest ponownie używany. Zamiast określenia współdzielenia parametrów można również powiedzieć, że sieć ma powiązane wagi (ang. *tied weights*), ponieważ wartość wagi zastosowanej do jednego wejścia jest powiązana z wartością wagi zastosowanej w innym miejscu. W sieci splotowej każdy element jądra jest używany w każdej pozycji wejścia (opcjonalnie poza pikselami na granicy obrazu).

Współdzielenie parametrów stosowanych w operacji splotu oznacza, że zamiast uczyć odzielnego zestawu parametrów dla każdej lokalizacji, wyznaczamy tylko jeden zestaw. Nie ma to wpływu na czas przebiegu w przód, wciąż wynosi $O(kn)$, ale dodatkowo zmniejsza wymagania pamięciowe modelu do k parametrów. Należy pamiętać, że k jest zazwyczaj kilka rzędów wielkości mniejsze niż m . Ponieważ m oraz n mają zwykle podobne rozmiary, k jest praktycznie pomijalne w porównaniu do $m \times n$. Splot pod względem wymagań pamięciowych i efektywności czasowej jest zatem znacznie bardziej efektywny niż mnożenie macierzy. Na rys. 3.5 zilustrowano współdzielenie parametrów w sieci splotowej.

W przypadku splotu współdzielenie parametrów powoduje, że warstwa ma właściwość nazywaną ekiwiariantnością (zob. [7, rozdz. 14]) ze względu na translacje (przesunięcia równoległe). Funkcja jest ekiwiariantna jeżeli w przypadku zmiany wejścia wyjście zmieni się w ten sam sposób.



Rysunek 3.4: Ilustracja warstw sieci splotowej z polami recepcyjnymi. Na rysunku widać, że przykładowe neurony w pierwszej warstwie splotowej (*Convolutional layer 1*) ograniczone są tylko do wycinka obrazu wejściowego (*Input layer*) tworząc pole recepcyjne, a neurony w drugiej warstwie splotowej (*Convolutional layer 2*) do wycinka pierwszej warstwy splotowej. Źródło: [7].

Definicja 3.5 (Ekwiariantność) Funkcja $f: \mathbb{R} \rightarrow \mathbb{R}$ jest *ekwiariantna* dla funkcji $g: \mathbb{R} \rightarrow \mathbb{R}$, jeśli

$$f(g(x)) = g(f(x)). \quad (3.5)$$

Jeśli w przypadku splotu przyjmiemy g jako dowolną funkcję, która przesuwa wejście, to funkcja splotu jest ekwiariantna dla g . Na przykład, niech I będzie funkcją określającą jasność obrazu w całkowitych współrzędnych. Niech g będzie funkcją, która przekształca jedną funkcję obrazu w inną funkcję obrazu, taką że $I' = g(I)$ jest funkcją obrazu, gdzie $I'(x, y) = I(x - 1, y)$. Odwzorowanie to przesuwa każdy piksel I o jednostkę w prawo. Jeśli zastosujemy to przekształcenie do I , a następnie wykonamy splot, wynik będzie taki sam, jakbyśmy zastosowali splot do I' , a następnie przekształcili wyjście.

Splot tworzy mapę dwuwymiarową, na której pokazane jest, gdzie występują określone cechy na wejściu. Mapa ta nazywana jest mapą cech (zob. podrozdz. 3.5). Jeśli przesuniemy obiekt na wejściu, jego reprezentacja przesunie się o taką samą odległość na wyjściu. Jest to przydatne, gdy wiemy, że pewna funkcja ma zastosowanie do małej liczby sąsiednich pikseli na wielu lokalizacjach wejścia.

Przy przetwarzaniu obrazu przydatne jest wykrywanie krawędzi w pierwszej warstwie sieci splotowej. Te same krawędzie pojawiają się w zasadzie wszędzie na obrazie, dlatego dobrze jest współdzielić parametry dla całego obrazu.

Jednak w niektórych przypadkach współdzielenie parametrów może nie być pożądane dla całego obrazu. Przykładowo, jeśli przetwarzamy obrazy, które są przyjęte w taki sposób, aby były skoncentrowane na twarzy jednej osoby, wówczas prawdopodobnie chcemy wyodrębnić różne cechy w różnych miejscach obrazu – część sieci przetwarzającej góre twarzy powinna szukać brwi, podczas gdy część sieci przetwarzającej dół twarzy powinna szukać nosa.

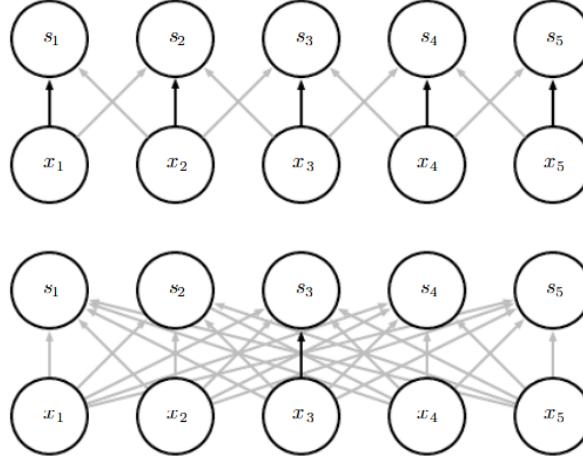
3.5 Filtr splotu i mapy cech

W tym podrozdziale przedstawiamy działanie filtru oraz wynik jego zastosowania – mapę cech. Niniejszy podrozdział powstał na podstawie [7, 9].

Przejdziemy teraz do definicji mapy cech.

Warstwa splotowa, która wykorzystuje jeden zestaw wag neuronów (jednoznacznie – jeden filtr splotu) w zastosowaniu do obrazu daje nam mapę cech (ang. *feature map*), która pozwala na dostrzeżenie elementów najbardziej przypominających dany filtr. Mapa cech jest zatem wynikiem warstwy splotowej. Mapa cech na wyjściu warstwy uzyskujemy tyle, ile filtrów zdefiniowaliśmy dla danej warstwy.

Wagi neuronów utożsamiane z filtrem splotu mogą być przedstawione jako obraz niewielkich rozmiarów równy rozmiarowi pola recepcyjnego. Na rys. 3.6 przedstawiono dwa różne zbiory wag – filtry. Pierwszy od lewej oznaczony jako czarny kwadrat z pionowym białym odcinkiem przechodzącym przez jego środek (możemy na niego patrzeć jako na macierz kwadratową, np. o wymiarze 5×5 , wypełnioną 0 poza środkową kolumną, w której mamy wartość 1). Tak skonstruowany filtr w przebiegu po obrazie zignoruje wszystkie elementy z pola recepcyjnego poza tymi znajdującymi się w środkowym pionowym odcinku, gdyż każdy element poza tym odcinkiem zostanie przemnożony przez 0. Drugi filtr od lewej oznaczony jest jako czarny kwadrat



Rysunek 3.5: Czarne strzałki na obrazku wskazują na połączenia, które korzystają z konkretnego parametru w dwóch różnych modelach. W modelu na górze rysunku czarne strzałki symbolizują „środkowy” element 3-elementowego jądra splotu w sieci splotowej. Z uwagi na współdzielenie parametrów, dany parametr wykorzystywany jest dla wszystkich neuronów wejściowych. W modelu u dołu rysunku czarna strzałka symbolizuje „środkowy” element macierzy wag w modelu klasycznej sieci neuronowej. W modelu tym nie występuje współdzielenie parametrów, wobec czego wskazany parametr wykorzystany jest tylko raz. Źródło: [9].

z poziomym białym odcinkiem przechodzącym przez jego środek. Podobnie dla tego przypadku pod uwagę brane są elementy leżące na odcinku poziomym.

W przypadku zastosowania do obrazu (a) na rysunku filtru „pionowego” uzyskamy efekt uwydatnienia pionowych linii przy jednoczesnym rozmyciu pozostałych elementów obrazu, co widoczne jest w obrazie (c) na rysunku. Odwrotna sytuacja ma miejsce w przypadku filtru „poziomego”, gdzie uwydatnione zostają linie poziome, a rozmyte pionowe, obserwujemy to w obrazie (b) na rysunku.

Podczas implementacji sieci splotowej nie ma potrzeby samodzielnie definiowania filtrów najbardziej przydatnych do danego zadania. Program inicjuje filtry o losowych wagach na początku uczenia, a następnie te wagi są dostosowywane podczas treningu sieci neuronowej, tzn. wykonywania algorytmu wstępnej propagacji błędu (opisanego w podrozdziale 2.6). Filtry te są macierzami wag, które przesuwane są po danych wejściowych podczas splotu, generując mapy cech dla kolejnych warstw sieci neuronowej.

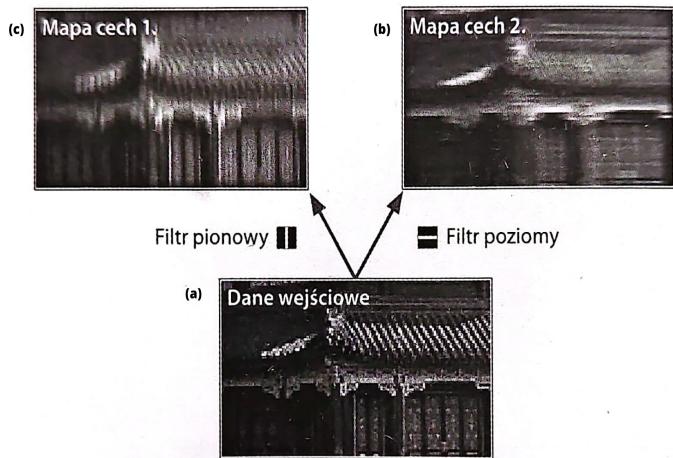
Dodatkowo należy podkreślić, że każdy filtr generuje jedną mapę cech. Oznacza to, że wynik zastosowania warstwy splotowej to mapy cech w liczbie równej filtrom, które dla danej warstwy zostały ustalone. Każda kolejna warstwa splotowa ma połączenie z każdą mapą cech, która jest wynikiem poprzedzającej jej warstwy splotowej (zob. [7, rozdz. 14]). W ten sposób zwiększa się „głębokość” danych wejściowych dla kolejnych warstw splotowych.

3.6 Warstwa łącząca

Niniejszy podrozdział traktuje o ważnym komponente architektury sieci splotowej, który zwiększa efektywność obliczeń, tzn. o warstwie łączącej. Podrozdział powstał na podstawie [7, 9].

Zasadniczym celem warstwy łączącej (ang. *pooling layer*) jest podpróbkowanie (ang. *subsampling*), lub inaczej zmniejszenie, obrazu na wejściu celem ograniczenia złożoności obliczeniowej i pamięciowej, a także ograniczenia liczby parametrów modelu, co naturalnie przekłada się na zmniejszenie ryzyka przetrenowania modelu (zob. podrozdział 1.5). Uszczegółowiąjąc, w tej warstwie wykorzystujemy funkcję łączącą (ang. *pooling function*), która zastępuje wyjście warstwy w pewnej lokalizacji miarą rozkładu wyjść w sąsiedztwie ustalonej lokalizacji. Lokalizację tę nazywa się jądrem łączącym (ang. *pooling kernel*). Istnieje wiele funkcji łączących, a najpopularniejszą jest maksymalizująca funkcja łącząca (ang. *max pooling function*). Zwraca ona największą wartość wyjścia w polu receptivejnym odpowiadającym neuronowi warstwy łączącej. Warstwa łącząca, która korzysta z tej funkcji, określana jest mianem maksymalizującej warstwy łączącej (ang. *max pooling layer*). Inne przykłady funkcji łączących to średnia wartości z pola receptivejnego i norma L^2 z pola receptivejnego.

Niezależnie od funkcji łączającej, warstwa łącząca pozwala na uzyskanie, w przybliżeniu, niezmienniczości (ang. *invariance*) ze względu na translację obrazu wejściowego. Oznacza to, że



Rysunek 3.6: Dwie mapy cech dla dwóch filtrów splotu. Filtr pionowy uwydatnia pionowe elementy zdjęcia i rozmywa te, które są pod innym kątem. Filtr poziomy uwydatnia poziome elementy zdjęcia i rozmywa te, które są pod innym kątem. Źródło: [7].

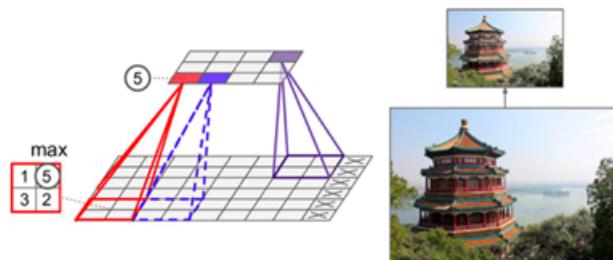
jeżeli przesuniemy obraz wejściowy o pewną niewielką wartość, to wartości wyjściowe warstwy łączącej nie zmienią się. Niezmienność ze względu na translacje może być bardzo korzystną właściwością, jeżeli interesuje nas bardziej fakt, że pewne cechy w ogóle występują na obrazie, niż to, gdzie dokładnie się znajdują. Na przykład w przypadku wykrywania twarzy na obrazie bardziej znaczące jest to, że po lewej stronie twarzy znajduje się oko, oraz że po prawej stronie znajduje się oko. Nie musimy znać ich lokalizacji z dokładnością do piksela, a jedynie istotny jest fakt ich obecności.

Po raz kolejny wart odnotowania jest fakt, że warstwa łącząca pozwala na znaczające zredukcowanie liczby parametrów przekazywanych do kolejnych warstw, co z kolei zwiększa wydajność obliczeniową i zmniejsza zapotrzebowanie pamięciowe modelu.

Z drugiej strony, warstwa łącząca może być także niebezpieczna. Przykładowo, maksymalizująca warstwa łącząca, nawet dla jądra o rozmiarach 2×2 , może zmniejszyć obraz wejściowy czterokrotnie, co może skutkować porzuceniem 75% obrazu wejściowego.

Dodatkowo, dla pewnych zadań, niezmienność ze względu na przesunięcie może być niepożądana. Szczególnie w przypadku zadania klasyfikacji każdego z pikseli obrazu wejściowego zgodnie z ich przynależnością do danego obiektu na obrazie, pożądana jest, aby po przesunięciu obrazu kilka pikseli w danym kierunku wynik również został przesunięty o kilka pikseli w tym samym kierunku.

Na rys. 3.7 przedstawiono ilustrację maksymalizującą warstwy łączącej.



Rysunek 3.7: Ilustracja maksymalizującej warstwy łączącej z jądrem łączącym o wymiarach 2×2 , która w konsekwencji prowadzi do zmniejszenia obrazu wejściowego. Źródło: [7].

3.7 Sieć splotowa w ujęciu formalnym

W poprzednich podrozdziałach przedstawiliśmy intuicyjne zrozumienie elementów sieci splotowej. W tym podrozdziale sformalizujemy matematycznie nasze rozważania. Struktura sieci splotowej podobna jest do struktury jednokierunkowej sieci neuronowej, w związku z tym formalizm matematyczny tych sieci jest podobny. Niniejszy podrozdział powstał w oparciu o [31, rozdz. 9].

Zaczniemy od zdefiniowania danych wejściowych, a więc tensora wejściowego wspomnianego w podrozdziale 3.3.

Definicja 3.6 (Tensor) *Tensorem wejściowym* (ang. *input tensor*) o wymiarze $q^{(1)} \times \dots \times q^{(K)}$ nazywamy wielowymiarową tablicę $\mathbf{x} \in \mathbb{R}^{q^{(1)} \times \dots \times q^{(K)}}$ rzędu $K \in \mathbb{N}$. Elementy tensora oznacza się przez $x_{i_1, \dots, i_K} \in \mathbb{R}$ dla $1 \leq i_k \leq q^{(k)}$ i $1 \leq k \leq K$.

Szczególny przypadek tensora rzędu $K = 2$ to macierz $\mathbf{x} \in \mathbb{R}^{q^{(1)} \times q^{(2)}}$. Macierz ta może być np. czarno-białym obrazem (tak jak na rys. 3.6) o wymiarach $q^{(1)} \times q^{(2)}$ z elementami macierzy $x_{i_1, i_2} \in \mathbb{R}$, które opisują intensywność szarości w odpowiadającym pikselu (i_1, i_2) .

Obraz kolorowy zazwyczaj ma trzy kanały koloru: czerwony (ang. *red*), zielony (ang. *green*) i niebieski (ang. *blue*). W skrócie kanały te określamy RGB. Obraz RGB może być reprezentowany przez tensor $\mathbf{x} \in \mathbb{R}^{q^{(1)} \times q^{(2)} \times q^{(3)}}$ rzędu 3, gdzie $q^{(1)} \times q^{(2)}$ określają wymiar obrazu, a $q^{(3)} = 3$ określa kanały RGB. Zatem $(x_{i_1, i_2, 1}, x_{i_1, i_2, 2}, x_{i_1, i_2, 3})^\top \in \mathbb{R}^3$ opisuje intensywność kolorów RGB w pikselu (i_1, i_2) .

Zazwyczaj strukturę obrazów czarno-białych i obrazów RGB ujednolica się, reprezentując obraz czarno-biały jako tensor $\mathbf{x} \in \mathbb{R}^{q^{(1)} \times q^{(2)} \times q^{(3)}}$ rzędu 3, z pojedynczym kanałem $q^{(3)} = 1$. To podejście będzie stosowane w całym tym podrozdziale. Mianowicie, jeśli rozważymy tensor $\mathbf{x} \in \mathbb{R}^{q^{(1)} \times \dots \times q^{(K-1)} \times q^{(K)}}$ rzędu K , pierwsze $K - 1$ składników (i_1, \dots, i_{K-1}) będzie odgrywać rolę elementów przestrzennych, a ostatnie składniki $1 \leq i_K \leq q^{(K)}$ nazywane są kanałami odzwierciedlającymi np. skalę szarości (dla $q^{(K)} = 1$) lub intensywności RGB (dla $q^{(K)} = 3$).

Rozpoczynamy od tensora wejściowego $\mathbf{x} \in \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}}$ rzędu K . Pierwsze $K - 1$ składniki tego tensora reprezentują strukturę przestrzenną, a K -ty składnik reprezentuje kanały. Warstwa splotowa stosuje (lokalne) operacje splotu na tym tensorze. Wybieramy rozmiar filtru $(f_m^{(1)}, \dots, f_m^{(K)})^\top \in \mathbb{N}^K$, gdzie $f_m^{(k)} \leq q_{m-1}^{(k)}$ dla $1 \leq k \leq K - 1$, oraz $f_m^{(K)} = q_{m-1}^{(K)}$. Filtry omawialiśmy w podrozdziale 3.5. Rozmiar filtru determinuje wymiar wyjściowy warstwy splotowej jako

$$q_m^{(k)} := q_{m-1}^{(k)} - f_m^{(k)} + 1, \quad (3.6)$$

dla $1 \leq k \leq K$. Stąd wymiar obrazu wejściowego jest zredukowany przez rozmiar filtru. W szczególności wymiar wyjściowy kanałów $k = K$ to $q_m^{(K)} = 1$, tzn. że wszystkie kanały są skompresowane w wyjściu do wartości skalarnej. Elementy przestrzenne $1 \leq k \leq K - 1$ zachowują swoją przestrzenną strukturę, ale wymiar jest zredukowany zgodnie ze wzorem (3.6).

Definicja 3.7 (Neuron warstwy splotowej) *Neuronem warstwy splotowej* nazywamy odwzorowanie $\mathbf{z}_j^{(m)} : \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \rightarrow \mathbb{R}^{q_m^{(1)} \times \dots \times q_m^{(K-1)}}$, takie że

$$\mathbf{z}_j^{(m)}(\mathbf{x}) = \left(z_{i_1, \dots, i_{K-1}; j}^{(m)}(\mathbf{x}) \right)_{1 \leq i_k \leq q_m^{(k)}; 1 \leq k \leq K-1}. \quad (3.7)$$

Pamiętajmy tu, że rząd tensora jest zredukowany z K do $K - 1$, bo kanały są kompresowane; indeks j w powyższym równaniu wyjaśnimy w dalszej części tekstu. Odwzorowanie (3.7) przyjmuje wartości dla ustalonej funkcji aktywacji $\phi : \mathbb{R} \rightarrow \mathbb{R}$

$$z_{i_1, \dots, i_{K-1}; j}^{(m)}(\mathbf{x}) = \phi \left(w_{0,j}^{(m)} + \sum_{l_1=1}^{f_m^{(1)}} \dots \sum_{l_K=1}^{f_m^{(K)}} w_{l_1, \dots, l_K; j}^{(m)} x_{i_1+l_1-1, \dots, i_{K-1}+l_{K-1}-1, l_K} \right), \quad (3.8)$$

dla danego członu obciążenia $w_{0,j}^{(m)} \in \mathbb{R}$ i wag

$$\mathbf{W}_j^{(m)} = \left(w_{l_1, \dots, l_K; j}^{(m)} \right)_{1 \leq l_k \leq f_m^{(k)}; 1 \leq k \leq K} \in \mathbb{R}^{f_m^{(1)} \times \dots \times f_m^{(K)}}. \quad (3.9)$$

Parametr sieci ma wymiar $r_m = 1 + \prod_{k=1}^K f_m^k$.

Na pierwszy rzut oka definicja ta może wydawać się nieco skomplikowana. Rozjaśnijmy zatem powyższy zapis i wprowadźmy bardziej zwięzłą notację. Operacja w (3.8) wybiera róg $(i_1, \dots, i_{K-1}, 1)$ jako punkt bazowy, a następnie odczytuje elementy tensora w polu recepcyjnym filtra

$$(i_1, \dots, i_{K-1}, 1) + [0 : f_m^{(1)} - 1] \times \dots \times [0 : f_m^{(K-1)} - 1] \times [0 : f_m^{(K)} - 1], \quad (3.10)$$

z danymi wagami $\mathbf{W}_j^{(m)}$. Pole recepcyjne jest następnie przesuwane po całym tensorze \mathbf{x} w wyniku zmiany punktu bazowego $(i_1, \dots, i_{K-1}, 1)$, ale przy zachowaniu ustalonych wag $\mathbf{W}_j^{(m)}$. Wyraża się tu cecha współdzielenia wag opisana w podrozdziale 3.4.

Dla $\mathbf{z}_j^{(m)} : \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \rightarrow \mathbb{R}^{q_m^{(1)} \times \dots \times q_m^{(K-1)}}$ neuron warstwy splotowej możemy zapisać w zwięzkiej notacji

$$\mathbf{z}_j^{(m)}(\mathbf{x}) = \phi \left(w_{0,j}^{(m)} + \mathbf{W}_j^{(m)} * \mathbf{x} \right), \quad (3.11)$$

gdzie mamy aktywację dla $1 \leq i_k \leq q_m^{(k)}$, $1 \leq k \leq K-1$,

$$\phi \left(w_{0,j}^{(m)} + \mathbf{W}_j^{(m)} * \mathbf{x} \right)_{i_1, \dots, i_{K-1}} = z_{i_1, \dots, i_{K-1}, k}^{(m)}(\mathbf{x}), \quad (3.12)$$

gdzie element $z_{i_1, \dots, i_{K-1}; j}^{(m)}(\mathbf{x})$ jest wyrażony przez (3.8). Operator $*$ symbolizuje korelację krzyżową omówioną w podrozdziale 3.3 (zob. def. 3.4) i nazywaną skrótnie splotem.

Przyglądając się powyższym równaniom, możemy zauważać analogie do jednokierunkowej sztucznej sieci neuronowej. W szczególności (3.11) pełni rolę neuronu w sieci splotowej w analogicznym sposobie, jak (2.7) pełni rolę neuronu w jednokierunkowej sieci neuronowej, w taki sposób, że $w_{0,j}^{(m)} + \mathbf{W}_j^{(m)} * \mathbf{x}$ zastępuje iloczyn skalarny $\langle \mathbf{w}_j^{(m)}, \mathbf{x} \rangle$. Dodatkowo neuron (2.7) może być rozpatrywany jako szczególny przypadek (3.11), jeżeli przyjmamy tensor rzędu $K=1$. Wówczas tensor wejściowy (czyli wektor) \mathbf{x} jest elementem przestrzeni $\mathbb{R}^{q_{m-1}^{(1)}}$. Mając taki tensor, nie mamy elementów przestrzennych, a tylko $q_{m-1} = q_{m-1}^{(1)}$ kanałów. W tym przypadku mamy $\mathbf{W}_j^{(m)} * \mathbf{x} = \langle \mathbf{W}_j^{(m)}, \mathbf{x} \rangle$ dla wag $\mathbf{W}_j^{(m)} \in \mathbb{R}^{q_{m-1}^{(1)}}$, gdzie zakładamy także brak członu obciążenia w \mathbf{x} . Zatem neuron w warstwie splotowej sprawdza się do warstwy w jednokierunkowej sieci w przypadku tensora rzędu 1.

W warstwie splotowej wykorzystujemy fakt, że tensor \mathbf{z} ma strukturę przestrzenną, cze- go nie ma w neuronie sieci jednokierunkowej. Neuron sieci splotowej przyjmuje wejście przestrzenne o wymiarze $\prod_{k=1}^K q_{m-1}^{(k)}$ i przekształca to wejście w przestrzenny obiekt o wymiarze $\prod_{k=1}^{K-1} q_m^{(k)}$. Neuron sieci splotowej korzysta z $r_m = 1 + \prod_{k=1}^K f_m^{(k)}$ wag. Neuron jednokierunkowej sieci przyjmuje wejście o wymiarze q_{m-1} i przekształca je w jednowymiarową aktywację neuronu, wykorzystując w tym celu $1 + q_{m-1}$ parametrów. Jeśli zidentyfikujemy wymiary wejścia jako $q_{m-1} = \prod_{k=1}^K q_{m-1}^{(k)}$, możemy zauważać, że $r_m \ll 1 + q_{m-1}$, ponieważ typowo rozmiary filtrów $f_m^{(k)} \ll q_{m-1}^{(k)}$ dla $1 \leq k \leq K$. Stąd neuron sieci splotowej wykorzystuje mniej parametrów, gdyż filtry działają tylko lokalnie poprzez operację splotu, przesuwając okno filtra (3.10), co omawiamy w podrozdziale 3.3 i 3.5.

Wszystko to pozwala teraz zdefiniować warstwę splotową sieci splotowej. Zauważmy, że odwzorowania (3.11) mają dolny indeks j , co wskazuje, że jest to pojedyncza projekcja. Wybierając wiele różnych wag $(w_{0,j}^{(m)}, \mathbf{W}_j^{(m)})$ (czyli różne filtry), możemy zdefiniować warstwę splotową w następujący sposób.

Definicja 3.8 (Warstwa splotowa) Wybierzmy $q_m^{(K)} \in \mathbb{N}$ filtry, każdy mający r_m -wymiarowe wagi $(w_{0,j}^{(m)}, \mathbf{W}_j^{(m)})$, $1 \leq j \leq q_m^{(K)}$. Warstwa splotowa to odwzorowanie $\mathbf{z}^{(m)} : \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \rightarrow \mathbb{R}^{q_m^{(1)} \times \dots \times q_m^{(K)}}$ zdefiniowane wzorem:

$$\mathbf{z}^{(m)}(\mathbf{x}) = (\mathbf{z}_1(\mathbf{x}), \dots, \mathbf{z}_{q_m^{(K)}}(\mathbf{x})), \quad (3.13)$$

z neuronami $\mathbf{z}_j^{(m)}(\mathbf{x}) \in \mathbb{R}^{q_m^{(1)} \times \dots \times q_m^{(K-1)}}, 1 \leq j \leq q_m^{(K)}$ danymi przez (3.11).

W przypadku rozpoznawaniu obrazu mamy tensor rzędu $K=3$. Zazwyczaj mamy obrazy o wymiarach (pixselach) $I \times J \times 3$ z trzema kanałami kolorów RGB. Obrazy te odczytywane są jako

$$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \in \mathbb{R}^{I \times J \times 3} = \mathbb{R}^{q_0^{(1)} \times q_0^{(2)} \times q_0^{(3)}}, \quad (3.14)$$

gdzie $\mathbf{x}_1 \in \mathbb{R}^{I \times J}$ to intensywność czerwonego, $\mathbf{x}_2 \in \mathbb{R}^{I \times J}$ to intensywność zielonego, a $\mathbf{x}_3 \in \mathbb{R}^{I \times J}$ to intensywność niebieskiego.

Definicja 3.9 (Warstwa splotowa dla kolorowego obrazu) Wybierzmy rozmiar pola receptivego $f_1^{(1)} \times f_1^{(2)}$ i $q_1 \in \mathbb{N}$ neuronów. Wówczas warstwa splotowa dla kolorowego obrazu to odwzorowanie $\mathbf{z}^{(1)} : \mathbb{R}^{I \times J \times 3} \rightarrow \mathbb{R}^{(I-f_1^{(1)})+1 \times (J-f_1^{(2)})+1 \times q_1}$, takie że

$$\mathbf{z}^{(1)}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = \left(\mathbf{z}_1^{(1)}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3), \dots, \mathbf{z}_{q_1}^{(1)}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \right), \quad (3.15)$$

z neuronami $\mathbf{z}_j^{(1)}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \in \mathbb{R}^{(I-f_1^{(1)})+1 \times (J-f_1^{(2)})+1 \times q_1}$, $1 \leq j \leq q_1$.

Zatem kompresujemy 3 kanały koloru w każdym neuronie j , ale zachowujemy strukturę przestrenną (dzięki operacji splotu).

Dla czarno-białych obrazów, które mają tylko jeden kanał koloru, zachowujemy strukturę przestrenną i modyfikujemy tensor wejściowy do tensora rzędu 3. Forma tensora wejściowego to

$$\mathbf{x} = (\mathbf{x}_1) \in \mathbb{R}^{I \times J \times 1}. \quad (3.16)$$

Do pełnego opisu sieci splotowej musimy sformalizować także pojęcia użyteczne w takiej sieci, o których po części już w tym rozdziale wspomnieliśmy.

Operacja (3.11) redukuje rozmiar wyjścia zgodnie z (3.6). Stąd jeżeli rozpoczniemy od obrazu o rozmiarach $100 \times 50 \times 1$ i jeżeli rozmiary filtru dane są przez $f_m^{(1)} = f_m^{(2)} = 9$, wówczas wyjście będzie rozmiaru $92 \times 42 \times q_1^{(3)}$. Czasem taka redukcja jest niepraktyczna, wówczas wykorzystujemy uzupełnianie tensora (ang. *padding*).

Definicja 3.10 (Uzupełnianie tensora) Uzupełnianie tensora $\mathbf{x} \in \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \times p_m^{(k)}$ parametrami, $1 \leq k \leq K-1$, to rozszerzenie tensora we wszystkich $K-1$ kierunkach przestrzennych poprzez dodawanie elementów (najczęściej zer), tak aby uzupełniony tensor miał wymiar

$$\left(p_m^{(1)} + q_{m-1}^{(1)} + p_m^{(1)} \right) \times \dots \times \left(p_m^{(K-1)} + q_{m-1}^{(K-1)} + p_m^{(K-1)} \right) \times q_{m-1}^{(K)}. \quad (3.17)$$

Wynika z tego, że neurony wyjściowe będą miały wymiar

$$q_m^{(k)} = q_{m-1}^{(k)} + 2p_m^{(k)} - f_m^{(k)} + 1, \quad (3.18)$$

dla $1 \leq k \leq K$. Uzupełnianie nie dodaje żadnych nowych parametrów, a jest jedynie wykorzystywane do zmiany kształtu tensora.

Kolejnym ważnym pojęciem jest krok (ang. *stride*). Kroki są wykorzystywane w celu ominięcia niektórych części wejściowego tensora \mathbf{x} w celu zredukowania rozmiaru wyjścia. Może być to użyteczne, jeżeli obraz wejściowy ma wysoką rozdzielczość.

Definicja 3.11 (Przesunięcia okna filtru z parametrami kroku) Ustalmy parametry kroku (nazywane też krokami) $s_m^{(k)}$, $1 \leq k \leq K-1$. Wówczas przesunięcia okna filtru z parametrami kroku to następująca operacja zamiany sumy

$$z_{i_1, \dots, i_{K-1}; j}^{(m)}(\mathbf{x}) = \phi \left(w_{0,j}^{(m)} + \sum_{l_1=1}^{f_m^{(1)}} \dots \sum_{l_K=1}^{f_m^{(K)}} w_{l_1, \dots, l_K; j}^{(m)} x_{s_m^{(1)}(i_1-1)+l_1, \dots, s_m^{(K-1)}(i_{K-1}-1)+l_{K-1}, l_K} \right). \quad (3.19)$$

Zauważmy, że jest to w rzeczywistości zastąpienie sumy z (3.8) z wykorzystaniem parametrów kroku.

Ta procedura wyodrębnia jedynie wartości tensora, które leżą na dyskretnej siatce tensora poprzez przesuwanie pola receptivego o wielokrotności liczb całkowitych,

$$\left(s_m^{(1)}(i_1-1), \dots, s_m^{(K-1)}(i_{K-1}-1), \right) + \left[0 : f_m^{(1)} \right] \times \dots \times \left[0 : f_m^{(K-1)} \right] \times \left[0 : f_m^{(K)} - 1 \right]. \quad (3.20)$$

Jeżeli wybierzemy kroki $s_m^{(k)} = f_m^{(k)}$, $1 \leq k \leq K-1$, otrzymujemy podział przestrzennej części tensora wejściowego \mathbf{z} , który będzie używany w warstwie łączającej.

Sformalizujmy teraz warstwę łączającą omówioną w podrozdziale 3.6.

Definicja 3.12 (Warstwa łącząca) Weźmy pole recepcyjne o ustalonym rozmiarze $(f_m^{(1)}, \dots, f_m^{(K-1)})^\top \in \mathbb{N}^{K-1}$ oraz kroki $s_m^{(k)} = f_m^{(k)}$, $1 \leq k \leq K-1$, dla elementów przestrzennych tensora \mathbf{x} rzędu K . Wówczas maksymalizująca warstwa łącząca $\mathbf{z}^{(m)} : \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \rightarrow \mathbb{R}^{q_m^{(1)} \times \dots \times q_m^{(K)}}$ dana jest przez

$$\mathbf{z}^{(m)}(\mathbf{x}) = \text{MaxPool}(\mathbf{x}), \quad (3.21)$$

z wymiarami $q_m^{(K)} = q_{m-1}^{(K)}$, a dla wymiarów $1 \leq k \leq K-1$ mamy

$$q_m^{(k)} = \lfloor q_{m-1}^{(k)} / f_m^{(k)} \rfloor. \quad (3.22)$$

Funkcja aktywacji dla $1 \leq i_k \leq q_m^{(k)}$, $1 \leq k \leq K-1$,

$$\text{MaxPool}(\mathbf{x})_{i_1, \dots, i_K} = \max_{1 \leq l_k \leq f_m^{(k)}, 1 \leq k \leq K-1} x_{f_m^{(1)}(i_1-1)+l_1, \dots, f_m^{(K-1)}(i_{K-1}-1)+l_{K-1}, i_K}. \quad (3.23)$$

To pozwala na wyodrębnienie wartości maksymalnych z pola recepcyjnego

$$\begin{aligned} & \left(f_m^{(1)}(i_1-1) + l_1, \dots, f_m^{(K-1)}(i_{K-1}-1) + l_{K-1}, i_K \right) + \left[1 : f_m^{(1)} \right] \times \dots \times \left[1 : f_m^{(K-1)} \right] \times [0] \\ &= \left[f_m^{(1)}(i_1-1) + 1 : f_m^{(1)} i_1 \right] \times \dots \times \left[f_m^{(K-1)}(i_{K-1}-1) + 1 : f_m^{(K-1)} i_{K-1} \right] \times [i_K], \end{aligned} \quad (3.24)$$

dla każdego kanału $1 \leq i_k \leq q_{m-1}^{(K)}$ osobno.

Operator max-pooling jest wybrany tak, aby wyodrębniał maksymalną wartość dla każdego kanału i każdego pola recepcyjnego, gdzie pola stanowią podział przestrzennej części tensora. To zmniejsza wymiar tensora zgodnie z równaniem (3.22). Przykładowo, jeżeli rozważymy tensor rzędu 3 obrazu RGB wymiaru $I \times J = 180 \times 50$ i zastosujemy maksymalizującą warstwę łączącą z polem recepcyjnym o rozmiarach $f_m^{(1)} = 10$ i $f_m^{(2)} = 5$ otrzymamy redukcję wymiaru

$$180 \times 50 \times 3 \rightarrow 18 \times 10 \times 3. \quad (3.25)$$

Warstwa spłaszczająca (ang. *flatten layer*) wykonuje transformację rozłożenia tensora na wektor, dzięki czemu wyjście warstwy spłaszczającej może być wykorzystane jako wejście do jednokierunkowej sieci.

Definicja 3.13 (Warstwa spłaszczająca) Warstwa spłaszczająca $\mathbf{z}^{(m)} : \mathbb{R}^{q_{m-1}^{(1)} \times \dots \times q_{m-1}^{(K)}} \rightarrow \mathbb{R}^{q_m}$ to

$$\mathbf{z}^{(m)}(\mathbf{x}) = \left(x_{1, \dots, 1}, \dots, x_{q_{m-1}^{(1)}, \dots, q_{m-1}^{(K)}} \right)^\top, \quad (3.26)$$

$$\text{z } q_m = \prod_{k=1}^K q_{m-1}^{(k)}.$$

3.8 Przykład architektury sieci splotowej

W tym podrozdziale przedstawiamy konkretny przykład architektury sieci splotowej. Podrozdział powstał na podstawie [31, rozdz. 9].

Założymy, że mamy obraz wejściowy RGB opisany przez tensor $\mathbf{x}^{(0)} \in \mathbb{R}^{I \times J \times 3}$ rzędu 3, który modeluje 3 kanały RGB obrazu ustalonego rozmiaru $I \times J$. Założymy ponadto, że mamy dane tabelaryczne cechy $\mathbf{x}^{(1)} \in \mathcal{X} \subset \{1\} \times \mathbb{R}^q$, które opisują dodatkowe właściwości danych. Oznacza to, że mamy zmienną wejściową w postaci $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)})$ i chcemy przewidzieć zmienną odpowiedzi Y poprzez wykorzystanie odpowiedniej funkcji regresji

$$\mu(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}) = \mathbb{E}[Y | \mathbf{x}^{(0)}, \mathbf{x}^{(1)}]. \quad (3.27)$$

Wybieramy dwie warstwy splotowe $\mathbf{z}^{(\text{CN1})}$ i $\mathbf{z}^{(\text{CN2})}$, każda z następującą po sobie maksymalizującą warstwą łączącą, odpowiednio $\mathbf{z}^{(\text{Max1})}$ i $\mathbf{z}^{(\text{Max2})}$. Następnie stosujemy warstwę spłaszczącą $\mathbf{z}^{(\text{flatten})}$ w celu uzyskania formy wektorowej wyuczonych reprezentacji danych. Warstwy te są wybrane zgodnie z opisem przedstawionym w poprzednim podrozdziale z dopasowaniem wymiarów wejścia i wyjścia, aby następujące złożenie było dobrze zdefiniowane.

$$\mathbf{z}^{(5:1)} = (\mathbf{z}^{(\text{flatten})} \circ \mathbf{z}^{(\text{Max2})} \circ \mathbf{z}^{(\text{CN2})} \circ \mathbf{z}^{(\text{Max1})} \circ \mathbf{z}^{(\text{CN1})}) : \mathbb{R}^{I \times J \times 3} \rightarrow \mathbb{R}^{q_5}. \quad (3.28)$$

Na listingu 3.1 widzimy przykład implementacji powyższych rozważań w Pythonie z wykorzystaniem biblioteki Keras. Tensor wejściowy $\mathbf{x}^{(0)}$ ma kształt $I \times J \times 3 = 180 \times 50 \times 3$ i uzyskujemy $q_5 = 60$ wymiarową wyuczoną reprezentację $\mathbf{z}^{(5:1)}(\mathbf{x}^{(0)}) \in \mathbb{R}^{60}$.

$$180 \times 50 \times 3 \xrightarrow{\text{CN1}} 170 \times 45 \times 10 \xrightarrow{\text{Max1}} 17 \times 9 \times 10 \xrightarrow{\text{CN1}} 12 \times 6 \times 5 \xrightarrow{\text{Max2}} 4 \times 3 \times 5 \xrightarrow{\text{flatten}} 60. \quad (3.29)$$

Pierwsza warstwa splotowa ($m = 1$) zawiera $q_1^{(3)} r_1 = 10 \cdot (1 + 11 \cdot 6 \cdot 3) = 1990$ wag $(w_{0,j}^{(1)}, \mathbf{W}_j^{(1)})_{1 \leq j \leq q_1^{(3)}}$ (zawierające także człony obciążenia). Druga warstwa splotowa ($m = 3$) zawiera $q_3^{(3)} r_3 = 5 \cdot (1 + 6 \cdot 4 \cdot 10) = 1205$ wag $(w_{0,j}^{(3)}, \mathbf{W}_j^{(3)})_{1 \leq j \leq q_3^{(3)}}$. Razem mamy parametr sieci o wymiarze 3195, który należy dopasować do tej architektury. Dopasowanie wag odbywa się za pomocą jednego z wariantów algorytmu gradientu prostego.

W celu wykonania zadania predykcji (3.27) łączymy wyuczoną reprezentację $\mathbf{z}^{(5:1)}(\mathbf{x}^{(0)}) \in \mathbb{R}^{q_5}$ obrazu RGB $\mathbf{x}^{(0)}$ z tabelarycznymi cechami $\mathbf{x}^{(1)} \in \mathcal{X} \subset \{1\} \times \mathbb{R}^q$. Taki wektor jest przetwarzany przez jednokierunkową sieć $\mathbf{z}^{(d+5:6)}$ o głębokości $d \geq 1$ i otrzymujemy wyjście

$$\mathbb{E} [\mathbf{Y} | \mathbf{x}^{(0)}, \mathbf{x}^{(1)}] = g^{-1} \left(\langle \beta, \mathbf{z}^{(d+5:6)} (\mathbf{z}^{(5:1)}(\mathbf{x}^{(0)}), \mathbf{x}^{(1)}) \rangle \right), \quad (3.30)$$

dla danej funkcji związku g .

```

1 from keras.models import Sequential
2 from keras.layers import Conv2D, MaxPooling2D, Flatten
3
4 # Creating model
5 model = Sequential()
6
7 # Adding layers to model
8 model.add(Conv2D(filters=10, kernel_size=(11, 6), activation='tanh', input_shape
9     =(180, 50, 3)))
10 model.add(MaxPooling2D(pool_size=(10, 5)))
11 model.add(Conv2D(filters=5, kernel_size=(6, 4), activation='tanh'))
12 model.add(MaxPooling2D(pool_size=(3, 2)))
13 model.add(Flatten())
14 # Show summary of model
15 model.summary()
```

Listing 3.1: Przykład implementacji architektury sieci splotowej w Pythonie z wykorzystaniem biblioteki Keras. Kod zmodyfikowany na podstawie listingu 9.1 z [31].

```

1 Model: "sequential"
2 -----
3 Layer (type)          Output Shape         Param #
4 -----
5 conv2d (Conv2D)        (None, 170, 45, 10)    1990
6 -----
7 max_pooling2d (MaxPooling2D) (None, 17, 9, 10)    0
8 -----
9 conv2d_1 (Conv2D)       (None, 12, 6, 5)      1205
10 -----
11 max_pooling2d_1 (MaxPooling2D) (None, 4, 3, 5)    0
12 -----
13 flatten (Flatten)      (None, 60)            0
14 -----
15 Total params: 3,195
16 Trainable params: 3,195
17 Non-trainable params: 0
18 -----
```

Listing 3.2: Wynik kodu z listingu 3.1. Widzimy tu kształty poszczególnych warstw i liczbę ich parametrów.

Rozdział 4

Zastosowanie uczenia głębokiego w analizie danych medycznych

W tym rozdziale przedstawiono część eksperymentalną pracy, w której stosujemy techniki przetwarzania obrazów medycznych z wykorzystaniem sztucznych sieci neuronowych. Bazować będziemy na teorii omawianej w rozdziale 2 i rozdziale 3. Na początku wykonujemy wybórowy przegląd wybranych przykładów zastosowań uczenia głębokiego dla danych medycznych we współczesnej literaturze. Eksplorację danych wykonujemy zgodnie z etapami procesu CRISP-DM opisanymi w podrozdziale 1.8. Architektury modeli i eksperymenty przeprowadzane w ramach modeli są podobne, z uwagi na ten fakt ich opisy dla każdego modelu w podrozdziały 4.6 i 4.7 celowo są do siebie zbliżone i różnią się szczegółami. Kolejne podrozdziały stanowią kolejne kroki procesu.

4.1 Przegląd literatury

W ostatnich latach, dzięki prejnemu rozwojowi technologii komputerowych, jesteśmy świadkami znacznego postępu w zastosowaniach uczenia głębokiego do różnych dziedzin nauki. Nie inaczej jest w przypadku medycyny. Przytoczymy kilka przykładów publikacji, które unaoczniają, jak szerokie spektrum zastosowań niesie ze sobą uczenie głębokie w analizie danych medycznych. W szczególności skupimy uwagę na technikach uczenia głębokiego w przetwarzaniu obrazu.

W artykule [8] badacze stworzyli model sieci splotowej, którego zadaniem była ocena złośliwości guzów nerek na podstawie zdjęcia tomografii komputerowej. Model osiągnął skuteczność na poziomie 87%. Do celów stworzenia autorskiego modelu predykcyjnego zdobyto ponad 15 tys. zdjęć tomografii komputerowej z niemal 400 przypadków medycznych. Sama architektura sieci składała się z 5 warstw splotowych (oraz 5 warstw łączących) i następującej po nich warstwie w pełni połączonej. Tensorem wejściowym były czarno-białe obrazy o wymiarach 130×130 .

W artykule [26] autorzy omawiają hybrydowy model klasyfikacyjny podtypów raka gruczołu sutkowego na obrazach histopatologicznych. Model łączy sieć splotową z rekurencyjną siecią neuronową i korzysta z uczenia transferowego. Sieć rekurencyjna to rodzaj sztucznej sieci neuronowej, która może analizować i przetwarzać ciągi danych, uwzględniając informacje z poprzednich kroków w celu generowania wyników. Uczenie transferowe (ang. *transfer learning*) to technika uczenia maszynowego, która wykorzystuje model wytrenowany na jednym zbiorze danych do usprawnienia osiągnięć w innym, podobnym zbiorze. Badanie oceniło model na zbiorze danych zawierającym obrazy raka gruczołu sutkowego o charakterze łagodnym i złośliwym. Hybrydowy model osiągnął wysoką dokładność wynoszącą 99% w przypadku klasyfikacji binarnej (łagodny vs. złośliwy) oraz 92,5% w przypadku klasyfikacji wieloklasowej (rozróżnianie różnych podtypów raka). Wyniki pokazują, że ta metoda uczenia transferowego przewyższa istniejące modele uczenia maszynowego i głębokiego uczenia, co czyni ją obiecującą metodą do klasyfikowania różnych rodzajów nowotworów i chorób.

W artykule [18] badacze przedstawiają zadanie klasyfikacji zdjęć histopatologicznych gruczolakora płuca z wykorzystaniem wielowarstwowego perceptronu, który został omówiony w podrozdziale 2.3. W szczególności autorzy stawiają sobie za cel określenie stopnia zaawansowania raka płuc począwszy od tkanki wolnej od nowotworu, przez niewielkie zmiany, do zaawansowanej infiltracji nowotworowej. Eksperymenty wykazały, że zaproponowany model, nazwany MIM, osiąga imponującą dokładność diagnozy wynoszącą 95,31% oraz wysoką precyzję, czułość i swoistość. MIM przewyższa wyniki innych modeli sieciowych, a także wykazuje skalowalność i stabilność. Wyniki sugerują, że MIM ma duży potencjał w zastosowaniach detekcji raka płuc, co może znacząco wpływać na skuteczność diagnozy tej choroby.

Dzięki przytoczonym artykułom widzimy, że zagadnienie rozpoznawania obrazów z wykorzystaniem technik uczenia głębokiego jest aktualnym tematem badań w świecie nauki. Możemy zaobserwować także mnogość technik wykorzystywanych w rozpoznawaniu obrazu, począwszy od klasycznych modeli, jak perceptrony wielowarstwowe, przez sieci splotowe, aż po zaawansowane modele hybrydowe, korzystające z uczenia transferowego.

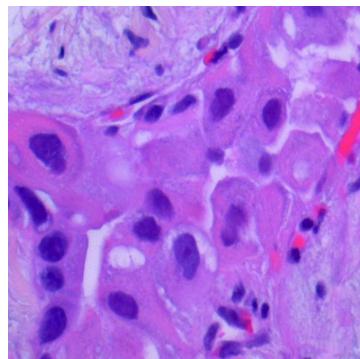
4.2 Opis zbioru danych

Wybrany zbiór danych pochodzi z serwisu Kaggle (zob. [17]) i został opublikowany w artykule [3]. Zawiera 25 000 obrazów histopatologicznych podzielonych na 5 klas. Wszystkie obrazy mają rozmiar 768×768 pikseli i są w formacie pliku jpeg. Obrazy zostały wygenerowane na podstawie oryginalnej próbki klinicznej ze źródeł zgodnych z normami HIPAA¹ przez lekarzy i badaczy ze szpitala *James A. Haley Veterans' Hospital*, Uniwersytetu Południowej Florydy oraz centrum badań onkologicznych *Moffitt Cancer Center*, co świadczy o rzetelności i wiarygodności danych.

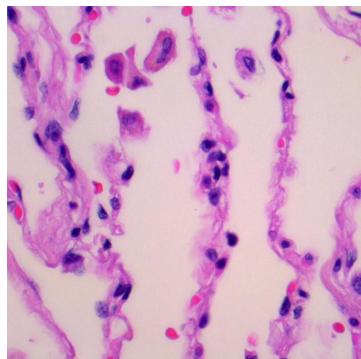
Oryginalna próbka kliniczna składała się z łącznie 750 obrazów tkanki płucnej (250 obrazów tkanki płucnej wolnej od nowotworu, 250 obrazów gruczolakoraka płuca i 250 obrazów raka płaskonabłonkowego płuca) oraz 500 obrazów tkanki okrężnicy (250 tkanki wolnej od nowotworu i 250 obrazów nowotworów złośliwych okrężnicy). Liczba obrazów została zwiększena do 25 000 za pomocą biblioteki *Augmentor* (zob. podrozdział 4.4).

Zestaw danych zawiera pięć klas, z każdą klasą składającą się z 5000 obrazów. Dla płuc są to

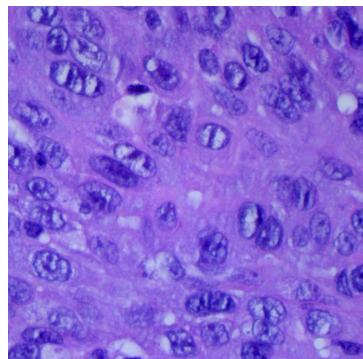
- Tkanka płucna wolna od nowotworu (ang. *benign lung tissue*).
- Gruczolakorak płuca (ang. *lung adenocarcinoma*).
- Rak płaskonabłonkowy płuca (ang. *lung squamous cell carcinoma*).



Rysunek 4.1: Gruczolakorak płuca. Źródło: zbiór danych [17].



Rysunek 4.2: Tkanka płucna wolna od nowotworu. Źródło: zbiór danych [17].

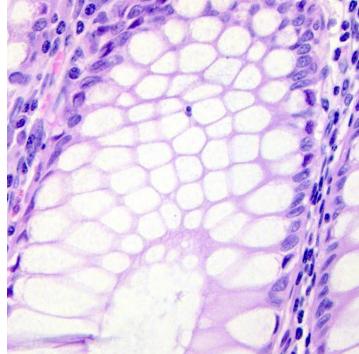


Rysunek 4.3: Rak płaskonabłonkowy płuca. Źródło: zbiór danych [17].

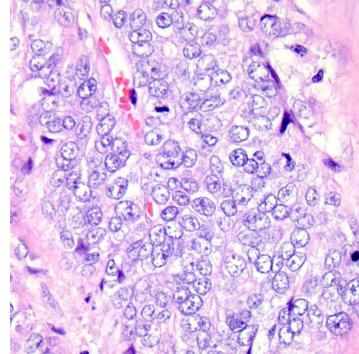
¹Ustawa *Health Insurance Portability and Accountability Act* (HIPAA) to profil zabezpieczeń skupiający się na ochronie danych EPHI (*Electronically Protected Health Information* – Elektronicznie chronione informacje dotyczące zdrowia).

Dla jelit:

- Tkanka okreżnicy wolna od nowotworu (ang. *colon benign tissue*).
- Gruczolakorak okreżnicy (ang. *colon adenocarcinoma*).



Rysunek 4.4: Tkanka okreżnicy wolna od nowotworu. Źródło: zbiór danych [17].



Rysunek 4.5: Gruczolakorak okreżnicy. Źródło: zbiór danych [17].

4.3 Rozpoznanie potrzeby badawczej

W naszych badaniach chcemy wykonywać klasyfikację tkanek zebranych obrazów. Ponieważ mamy do czynienia z dwoma rodzajami narządów wewnętrznych, płucami oraz jelitami, skonstruujemy dwa klasyfikatory.

- Zadaniem pierwszego będzie klasyfikacja obrazów histopatologicznych jelita grubego na dwie możliwe klasy. W celu rozpoznania tkanek zdrowych i tkanek chorych. Będzie to przypadek klasyfikacji binarnej.
- Zadaniem drugiego będzie klasyfikacja dla obrazów tkanki płucnej. Będzie to przypadek klasyfikacji wieloklasowej z uwagi na istnienie trzech klas obrazów.

Wyzwania, z jakimi się musimy zmierzyć, to przede wszystkim ograniczenia komputera związane ze złożonością obliczeniową algorytmów treningu sieci. Z tego powodu kod uruchamiamy bieżącym na chmurze serwisu *Kaggle*.

Klasyfikatory, które stworzymy, będą modelami sieci splotowej. Zbudujemy po 2 modele o różnych architekturach dla obrazów tkanki płucnej i obrazów tkanki jelita. Porównamy je pod kątem jakości i efektywności, celem wyboru najlepszego klasyfikatora dla naszego zadania.

4.4 Zrozumienie danych

Zbiór danych, nad którym będziemy pracować, został dostarczony w artykule [3]. Według wyjaśnień autorów tego artykułu, zbiór został stworzony w odpowiedzi na potrzebę poszerzenia dostępności zestawów danych obrazowych w celach badawczych, ze szczególnym uwzględnieniem zapotrzebowania na obszerne zbiory danych treningowych dla modeli uczenia maszynowego.

Autorzy artykułu dokonali istotnych przekształceń danych, które umożliwiają ich efektywne wykorzystanie w modelach. W szczególności, wszystkie obrazy zostały przycięte do formatu kwadratowego o rozmiarze 768×768 pikseli, uzyskanego poprzez przeskalowanie pierwotnych wymiarów 1024×768 pikseli.

Kolejnym istotnym krokiem było poddanie obrazów augmentacji. Augmentacja danych to technika przetwarzania danych stosowana w uczeniu maszynowym, a szczególnie w uczeniu głębokim, polegająca na generowaniu nowych przykładów danych poprzez wprowadzanie różnorodnych, kontrolowanych zmian do istniejących danych treningowych, takich jak np. obracanie czy przesuwanie obrazów. Bardziej szczegółowy opis procesu augmentacji można znaleźć w [9, rozdz. 7]. Do tego celu użyto narzędzia oprogramowania o nazwie *Augmentor*. *Augmentor* to biblioteka napisana w języku Python, specjalnie zaprojektowana z myślą o augmentacji obrazów w kontekście uczenia maszynowego. Dzięki zastosowaniu *Augmentor*, zbiór danych został powiększony

do 25 000 zdjęć poprzez zastosowanie różnorodnych operacji augmentacji, takich jak rotacje w lewo i prawo (w zakresie do 25 stopni) oraz odbicia poziome i pionowe. Warto zaznaczyć, że dla każdego obrazu zachowana została odpowiednia etykieta.

Wszystkie te działania skutkują tym, że zbiór danych nie wymaga dodatkowej obróbki w celu oczyszczenia i wzbogacenia. Zbiór ten jest więc gotowy do wykorzystania w naszym programie, co pozwoli na skuteczne przeprowadzenie procesu analizy i modelowania.

4.5 Przygotowanie danych

Z uwagi na wyjątkowo wysoką jakość dostępnych danych, istnieje niewielka potrzeba przeprowadzania procesu oczyszczania danych. Dane w zbiorze są już na tyle rzetelne i „czyste”, że nie wymagają dodatkowych operacji usuwania lub korekcji anomalii, czy augmentacji. Jedynym wykonanym przez nas przekształceniem dostępnych danych jest zmiana rozmiaru obrazów z 768×768 na 224×224 , w celu obniżenia złożoności obliczeniowej i skrócenia czasu treningu. Format 224×224 jest powszechnie używanym rozmiarem wejściowym dla wielu modeli sieci spłotowych, a niekiedy przez nie wymaganym (zob. dokumentację modelu ResNet [22]).

W celu efektywnego zarządzania i analizy dostępnych danych, przeprowadzamy podział zbioru na dwa główne konteksty:

- Zbiór danych dotyczący jelita (2 klasy).
- Zbiór danych dotyczący płuc (3 klasy).

Każdy z tych zbiorów skupia się na innych typach obrazów i posiada inną liczbę klas, co jest istotne dla procesu klasyfikacji.

Następnie każdy z tych zbiorów jest podzielony na bardziej szczegółowe podzbiory zgodnie z klasami tkanek, które są opisane w podrozdziale 4.2. Dla każdej z tych klas zostaje przypisana odpowiednia etykieta tkanki, co pozwoli na właściwą identyfikację podczas procesu uczenia modelu:

- Tkanka płuca wolna od nowotworu \Rightarrow etykieta: *Lung Benign Tissue*.
- Gruczolakorak płuca \Rightarrow etykieta: *Lung Adenocarcinoma*.
- Rak płaskonabłonkowy płuca \Rightarrow etykieta: *Lung Squamous Cell Carcinoma*.
- Tkanka okrężnicy wolna od nowotworu \Rightarrow etykieta: *Colon Benign Tissue*.
- Gruczolakorak rak okrężnicy \Rightarrow etykieta: *Colon Adenocarcinoma*.

Przeprowadzamy dalszy podział każdego z utworzonych podzbiorów na trzy kluczowe części: zbiór treningowy, zbiór walidacyjny oraz zbiór testowy. Zastosowanie proporcji 80% dla zbioru treningowego oraz po 10% dla zbiorów walidacyjnego i testowego pozwala na zachowanie równowagi pomiędzy liczbą przykładów używanych do trenowania, walidacji i testowania modelu.

Aby efektywnie wykorzystać dostępne dane podczas procesu uczenia, przekazujemy je do modelu w formie partii (ang. *batch*) po 128 przykładów. Taki rozmiar partii został wybrany z uwagi na kompromis pomiędzy szybkością przetwarzania a wykorzystaniem zasobów obliczeniowych.

Zastosowane przekształcenia wynikają z rozpoznania istotności procesu podziału danych na różne konteksty, klasy tkanek oraz zestawy treningowe, walidacyjne i testowe w celu optymalizacji efektywności treningu i oceny modelu. Dodatkowo, właściwy dobór rozmiarów partii ma wpływ na szybkość treningu, jednocześnie zachowując jakość osiąganych wyników. Trening z niewielkimi partiami zawierającymi tylko kilka przykładów uczących może prowadzić do wydłużonego czasu trwania procesu uczenia, ze względu na częstsze aktualizacje wag modelu. Jednak korzysta z mniejszej ilości pamięci RAM, co jest korzystne na komputerach z ograniczoną ilością dostępnej pamięci. W przypadku dużych partii, które zawierają więcej przykładów uczących, proces uczenia może zostać przyspieszony, ponieważ aktualizacje wag są wykonywane rzadziej, co prowadzi do lepszego wykorzystania obliczeń. To z kolei może skrócić czas trwania pojedynczej epoki treningu. Niemniej jednak, trenowanie z dużymi partiami wymaga większej ilości pamięci.

4.6 Budowa modeli

Modele, które zostaną utworzone dla przypadków związanych z płucami i jelitami, będą miały podobne architektury, z niewielkimi różnicami w ostatniej warstwie, która odpowiada za dokonywanie predykcji. Oznacza to, że nasze modele będą miały te same lub podobne warstwy splotowe i warstwy łączące w celu ekstrakcji cech z obrazów tkanki płucnej i obrazów tkanki jelita. Ostatnia warstwa modelu zależy od konkretnego zadania klasyfikacji. Dla jelit to klasyfikacja na zdrowe i chore tkanki, podczas gdy dla płuc to klasyfikacja różnych rodzajów schorzeń płuc i przypadków zdrowych.

Zbadamy, jak modele zachowują się w przypadku różnej liczby klas. To pozwoli nam zrozumieć, jakie wyzwanie i różnice mogą wystąpić w procesie uczenia maszynowego, gdy mamy do czynienia z różnymi rodzajami klasyfikacji medycznych. Daje to również możliwość dostosowania modelu do konkretnego przypadku z zachowaniem ogólnego schematu architektury.

Model 1 dla obrazów tkanki jelita

Zaczynamy od modeli dla obrazów tkanki jelita.

Pierwszy model, który wykorzystamy, to sieć splotowa o następującej architekturze (zapis zgodnie z konwencją podaną w podrozdziale 3.8)

$$\mathbf{z}^{(14:1)}(\mathbf{x}) = (\mathbf{z}^{(\text{Dense3})} \circ \mathbf{z}^{(\text{Dropout2})} \circ \mathbf{z}^{(\text{Dense2})} \circ \mathbf{z}^{(\text{Dropout1})} \circ \mathbf{z}^{(\text{Dense1})} \circ \mathbf{z}^{(\text{Flatten})} \circ \mathbf{z}^{(\text{Max4})} \circ \mathbf{z}^{(\text{CN4})} \circ \mathbf{z}^{(\text{Max3})} \circ \mathbf{z}^{(\text{CN3})} \circ \mathbf{z}^{(\text{Max2})} \circ \mathbf{z}^{(\text{CN2})} \circ \mathbf{z}^{(\text{Max1})} \circ \mathbf{z}^{(\text{CN1})})(\mathbf{x}). \quad (4.1)$$

Przedstawiamy kolejne warstwy występujące w architekturze sieci.

1. Warstwa splotowa 1 ($\mathbf{z}^{(\text{CN1})}$):

- Liczba filtrów: 64
- Rozmiar filtra: 3×3
- Funkcja aktywacji: ReLU
- Uzupełnianie: zerami

2. Maksymalizująca warstwa łącząca 1 ($\mathbf{z}^{(\text{Max1})}$):

- Rozmiar filtra: 2×2

3. Warstwa splotowa 2 ($\mathbf{z}^{(\text{CN2})}$):

- Liczba filtrów: 128
- Rozmiar filtra: 3×3
- Funkcja aktywacji: ReLU
- Uzupełnianie: zerami

4. Maksymalizująca warstwa łącząca 2 ($\mathbf{z}^{(\text{Max2})}$):

- Rozmiar filtra: 2×2

5. Warstwa splotowa 3 ($\mathbf{z}^{(\text{CN3})}$):

- Liczba filtrów: 256
- Rozmiar filtra: 3×3
- Funkcja aktywacji: ReLU
- Uzupełnianie: zerami

6. Maksymalizująca warstwa łącząca 3 ($\mathbf{z}^{(\text{Max3})}$):

- Rozmiar filtra: 2×2

7. Warstwa splotowa 4 ($\mathbf{z}^{(\text{CN4})}$):

- Liczba filtrów: 512
- Rozmiar filtra: 3×3
- Funkcja aktywacji: ReLU
- Uzupełnianie: zerami

8. Maksymalizująca warstwa łącząca 4 ($\mathbf{z}^{(\text{Max4})}$):

- Rozmiar filtra: 2×2

9. Warstwa spłaszczająca ($\mathbf{z}^{(\text{Flatten})}$)

10. Warstwa w pełni połączona 1 ($\mathbf{z}^{(\text{Dense1})}$):

- Liczba neuronów: 512
- Funkcja aktywacji: ReLU

11. Warstwa porzucenia 1 ($\mathbf{z}^{(\text{Dropout1})}$):

- Współczynnik porzucenia: 0,5

12. Warstwa w pełni połączona 2 ($\mathbf{z}^{(\text{Dense2})}$):

- Liczba neuronów: 256
- Funkcja aktywacji: ReLU

13. Warstwa porzucenia 2 ($\mathbf{z}^{(\text{Dropout2})}$):

- Współczynnik porzucenia: 0,5

14. Warstwa wyjściowa ($\mathbf{z}^{(\text{Dense3})}$)

- Liczba neuronów: 1
- Funkcja aktywacji: Logistyczna

Warstwę splotową przedstawiliśmy w definicji 3.9 i podrozdziale 3.4, warstwę łączącą w definicji 3.12 i podrozdziale 3.6, warstwę spłaszczącą w definicji 3.13, warstwę w pełni połączoną w definicji 2.1, a warstwę porzucenia w podrozdziale 2.4. Filtry, które przyjmuje sieć są dobierane w sposób automatyczny przez metodę *Conv2D* w Pythonie, odpowiadającą za warstwę sieci i nie są one nam znane, o czym mówiliśmy w podrozdziale 3.5. Funkcje aktywacji przedstawiono w podrozdziale 3.7.

Z architektury sieci wynika, że jej parametr ma wymiar 53 063 297. Jest to liczba wag, którą algorytm treningowy będzie optymalizował. Liczba epok, którą ustalamy, to 30.

Do optymalizacji wykorzystujemy algorytm *Adamax*². Jako funkcję straty dewiancji przyjmujemy binarną entropię krzyżową (zob. definicja 1.8).

Nasz tensor wejściowy \mathbf{x} ma wymiar $224 \times 224 \times 3$ z uwagi na fakt, iż jest to kwadratowy obraz RGB o wymiarze 224×224 pikseli. Będzie tak dla każdego stworzonego przez nas modelu.

Model 2 dla obrazów tkanki jelita

Model 2 dla obrazów tkanki jelita ma następującą architekturę

$$\begin{aligned} \mathbf{z}^{(16:1)}(\mathbf{x}) = & (\mathbf{z}^{(\text{Dense3})} \circ \mathbf{z}^{(\text{Dropout2})} \circ \mathbf{z}^{(\text{Dense2})} \circ \mathbf{z}^{(\text{Dropout1})} \circ \mathbf{z}^{(\text{Dense1})} \circ \mathbf{z}^{(\text{Flatten})} \circ \mathbf{z}^{(\text{Max5})} \circ \\ & \mathbf{z}^{(\text{CN5})} \circ \mathbf{z}^{(\text{Max4})} \circ \mathbf{z}^{(\text{CN4})} \circ \mathbf{z}^{(\text{Max3})} \circ \mathbf{z}^{(\text{CN3})} \circ \mathbf{z}^{(\text{Max2})} \circ \mathbf{z}^{(\text{CN2})} \circ \mathbf{z}^{(\text{Max1})} \circ \mathbf{z}^{(\text{CN1})})(\mathbf{x}). \end{aligned} \quad (4.2)$$

²Adamax jest bardziej rozbudowanym wariantem gradientu prostego, który wykorzystuje ruchome średnie, aby regulować tempo uczenia się i skompensować problemy związane z niespójnością skali gradientów w różnych wagach sieci neuronowej. Po raz pierwszy został opublikowany w [13].

Przedstawiamy kolejne warstwy występujące w architekturze sieci.

1. Warstwa splotowa 1 ($z^{(CN1)}$):

- Filtry: 64
- Rozmiar filtru: 3×3
- Funkcja aktywacji: Tangens hiperboliczny
- Uzupełnianie: zerami

2. Maksymalizująca warstwa łącząca 1 ($z^{(Max1)}$):

- Rozmiar filtru: 2×2

3. Warstwa splotowa 2 ($z^{(CN2)}$):

- Filtry: 128
- Rozmiar filtru: 3×3
- Funkcja aktywacji: Tangens hiperboliczny
- Uzupełnianie: zerami

4. Maksymalizująca warstwa łącząca 2 ($z^{(Max2)}$):

- Rozmiar filtru: 2×2

5. Warstwa splotowa 3 ($z^{(CN3)}$):

- Filtry: 256
- Rozmiar filtru: 3×3
- Funkcja aktywacji: Tangens hiperboliczny
- Uzupełnianie: zerami

6. Maksymalizująca warstwa łącząca 3 ($z^{(Max3)}$):

- Rozmiar filtru: 2×2

7. Warstwa splotowa 4 ($z^{(CN4)}$):

- Filtry: 512
- Rozmiar filtru: 3×3
- Funkcja aktywacji: Tangens hiperboliczny
- Uzupełnianie: zerami

8. Maksymalizująca warstwa łącząca 4 ($z^{(Max4)}$):

- Rozmiar filtru: 2×2

9. Warstwa splotowa 5 ($z^{(CN5)}$):

- Filtry: 700
- Rozmiar filtru: 3×3
- Funkcja aktywacji: Tangens hiperboliczny
- Uzupełnianie: zerami

10. Maksymalizująca warstwa łącząca 5 ($z^{(Max5)}$):

- Rozmiar filtru: 2×2

11. Warstwa spłaszczająca ($z^{(Flatten)}$)

12. Warstwa w pełni połączona 1 ($z^{(\text{Dense1})}$):

- Liczba neuronów: 512
- Funkcja aktywacji: ReLU

13. Warstwa porzucenia 1 ($z^{(\text{Dropout1})}$):

- Współczynnik porzucenia: 0,5

14. Warstwa w pełni połączona 2 ($z^{(\text{Dense2})}$):

- Liczba neuronów: 256
- Funkcja aktywacji: ReLU

15. Warstwa porzucenia 2 ($z^{(\text{Dropout2})}$):

- Współczynnik porzucenia: 0,5

16. Warstwa wyjściowa ($z^{(\text{Dense3})}$)

- Liczba neuronów: 1
- Funkcja aktywacji: Logistyczna

Z architektury sieci wynika, że jej parametr ma wymiar 22 470 973. Jest to zatem liczba wag, którą algorytm treningowy będzie optymalizował. Liczba epok, którą ustalamy to 30.

Do optymalizacji wykorzystujemy algorytm stochastycznego gradientu prostego omawiany w podrozdziale 2.6. Jako funkcję straty dewiancji przyjmujemy binarną entropię krzyżową.

Model 1 dla obrazów tkanki płucnej

Przechodzimy do modeli dla obrazów tkanki płucnej.

Pierwszy model dla obrazów tkanki płucnej jest analogiczny dla modelu (4.1), z różnicą w warstwie wyjściowej z uwagi na klasyfikację wieloklasową.

14. Warstwa wyjściowa ($z^{(\text{Dense3})}$)

- Liczba neuronów: 3
- Funkcja aktywacji: Softmax

Parametr takiej sieci ma wymiar 53 063 811.

Jako algorytm treningowy wybieramy algorytm *Adamax*. Jako funkcję straty dewiancji przyjmujemy kategorialną entropię krzyżową (zob. definicja 1.9).

Model 2 dla obrazów tkanki płucnej

Drugi model dla obrazów tkanki płucnej jest analogiczny dla modelu (4.2), z różnicą w warstwie wyjściowej z uwagi na klasyfikację wieloklasową.

16. Warstwa wyjściowa ($z^{(\text{Dense3})}$)

- Liczba neuronów: 3
- Funkcja aktywacji: Softmax

Parametr takiej sieci ma wymiar 22 471 487.

Jako algorytm treningowy wybieramy algorytm stochastycznego gradientu prostego. Jako funkcję straty dewiancji przyjmujemy kategorialną entropię krzyżową.

Motywacje

Omówimy teraz motywacje, które przyświecały nam przy konstrukcji architektury modelu 1 i modelu 2. Nie rozróżniamy tu modeli względem zbioru danych, z uwagi na jedyną różnicę w warstwie wyjściowej w zależności od modelu dla zbioru danych. Podkreślmy w tym miejscu, że nie ma ogólnie przyjętych zasad i „złotych standardów” co do konstrukcji architektur sieci. Pomysły na architektury sieci najczęściej pochodzą z rozpatrywania podobnych architektur dla zadań i zbiorów danych, z jakimi możemy mieć do czynienia sami.

W przypadku modelu 1 liczba warstw splotowych została dobrana w celu znalezienia kompromisu pomiędzy czasem treningu a efektywnością modelu. W ogólności większa liczba warstw powinna zwiększać poprawność modelu, jednak niesie to ze sobą ryzyko przetrenowania oraz znacznie zwiększa czas treningu. Po przeprowadzeniu wstępnych eksperymentów, 4 warstwy splotowe zdają się być dobrym kompromisem pomiędzy złożonością a efektywnością.

Funkcja aktywacji ReLu przyjęta dla modelu 1, w praktyce uważana jest za najlepszą funkcję aktywacji (zob. [7, rozdz. 10]) dla sieci neuronowych. W celu osiągnięcia najlepszych efektów uczenia, zdecydowaliśmy się na ten wybór.

Liczba filtrów w warstwach splotowych została przyjęta przez wyszukiwanie najlepszych praktyk dla sieci splotowych na różnych stronach internetowych zrzeszających specjalistów z dziedziny uczenia maszynowego (zob. np. [4]) oraz przykładów z książek (zob. [7, rozdz. 14]). Ponownie należy podkreślić, że nie istnieje ogólnie przyjęta najlepsza liczba filtrów dla sieci splotowych oraz neuronów dla sieci jednokierunkowych. Liczba ta różni się w zależności od zadania oraz nie znamy analitycznych technik na jej optymalizację, a sprawdzanie metodą „prób i błędów” jest zbyt czasochłonne, biorąc pod uwagę czas treningu nawet najprostszych sieci.

Algorytm treningu Adamax został wybrany poprzez eksperyment z porównaniem algorytmu stochastycznego gradientu prostego, przy którym dokładność zatrzymała się na poziomie ok. 50% we wszystkich epokach treningu.

Architektura modelu 2 została wybrana niejako w celu skonfrontowania niektórych wyborów przyjętych dla modelu 1. I tak, wybierając funkcję aktywacji tangensa hiperbolicznego spodziewamy się mniejszej wydajności modelu, więc dodajemy kolejną warstwę splotową, która w teorii powinna ją zwiększyć. Jako algorytm treningu wybieramy algorytm stochastycznego gradientu prostego i odrzucamy algorytm Adamax, z tej samej przyczyny, z której postąpiliśmy odwrotnie w przypadku modelu 1.

4.7 Ocena modeli pod kątem jakości i efektywności

W niniejszym podrozdziale wykonujemy ocenę stworzonych przez nas modeli, które przedstawiłyśmy w podrozdziale 4.6. W tym miejscu przypomnijmy, że układ podrozdziału przedstawia modele po kolej, co implikuje pewną powtarzalność w opisie kolejnych wyników. Należy zatem mieć na uwadze, że jest to zabieg celowy i wynikający z formy opisu. Na koniec podsumowujemy wszystkie modele oraz wyciągamy wnioski o ich wynikach, odnosząc się przy tym do ich budowy. W końcowej części tego podrozdziału dokonujemy podsumowania wszystkich modeli oraz wyciągamy wnioski na temat ich wyników, odnosząc się jednocześnie do ich struktury. Dodatkowo przeprowadzamy eksperyment polegający na trenowaniu sieci neuronowej bez warstw porzucenia i porównujemy jej wyniki z modelem zawierającym te warstwy. Jako drugi eksperyment sprawdzamy zachowanie się modelu 1 dla obrazów tkanki jelita przy zastosowaniu tangensa hiperbolicznego jako funkcję aktywacji w warstwach splotowych.

Model 1 dla obrazów tkanki jelita

Zaczynamy od modeli dla obrazów tkanki jelita.

Przebieg treningu modelu 1, reprezentowanego przez (4.1), przedstawiono w tabeli 4.1.

Tabela 4.1: Przebieg treningu dla modelu 1 dla obrazów tkanki jelita. Przez (T) oznaczono wartości dla danych treningowych, a przez (W) dla danych walidacyjnych.

Epoka	Funkcja straty (T)	Dokładność (T)	Funkcja straty (W)	Dokładność (W)
1	7,3319	0,5530	0,6643	0,5530
2	0,5950	0,6777	0,4895	0,7590
3	0,5770	0,6799	0,6303	0,5510
4	0,4764	0,7675	0,3507	0,8560
5	0,4359	0,7912	0,4256	0,7870
6	0,3728	0,8322	0,3817	0,8060
7	0,2458	0,8964	0,1480	0,9380
8	0,3342	0,8511	0,1979	0,9270
9	0,1694	0,9317	0,1113	0,9570
10	0,1437	0,9438	0,1152	0,9510
11	0,1318	0,9481	0,1661	0,9330
12	0,1148	0,9564	0,1775	0,9210
13	0,0899	0,9661	0,0755	0,9680
14	0,0553	0,9811	0,0737	0,9730
15	0,0226	0,9919	0,0764	0,9760
16	0,0228	0,9923	0,2097	0,9410
17	0,0150	0,9961	0,0583	0,9850
18	0,1970	0,9301	0,0847	0,9700
19	0,0283	0,9919	0,0705	0,9810
20	0,0153	0,9955	0,0750	0,9800
21	0,0087	0,9979	0,0684	0,9760
22	0,0082	0,9979	0,1653	0,9630
23	0,0124	0,9967	0,0764	0,9810
24	0,0039	0,9991	0,0570	0,9830
25	0,0026	0,9995	0,0703	0,9830
26	0,0030	0,9992	0,0792	0,9820
27	0,0037	0,9992	0,0587	0,9840
28	0,0032	0,9994	0,0996	0,9830
29	0,0108	0,9965	0,0679	0,9820
30	0,0111	0,9959	0,1554	0,9620

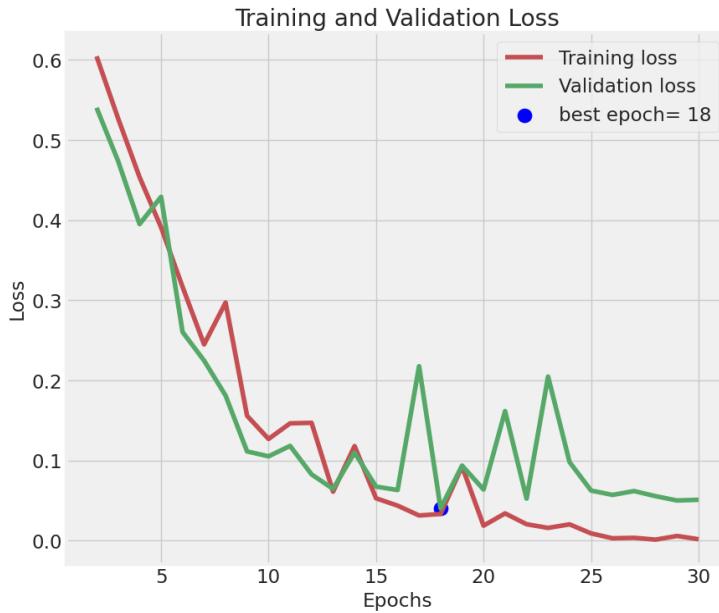
Jako metrykę poprawności modelu wybraliśmy dokładność (ang. *accuracy*). Dokładność mierzy stosunek liczby poprawnie sklasyfikowanych obiektów do ogólnej liczby obiektów w zbiorze danych. Metrykę tę stosować będziemy w treningach wszystkich stworzonych modeli. Wybór tej metryki jest uzasadniony z uwagi na zrównoważenie przypadków dla każdej z klasy (1:1 dla obrazów tkanki jelita i 1:1:1 dla obrazów tkanki płucnej), w przeciwnym przypadku należałoby skorzystać z innych metryk np. wskaźnika F1.

W pierwszych epokach treningu model wykazywał ograniczoną skuteczność. Wartość funkcji straty była podwyższona, a dokładność klasyfikacji utrzymywała się poniżej 90%. To sygnalizuje, że model napotykał pewne trudności w dokładnym identyfikowaniu obiektów.

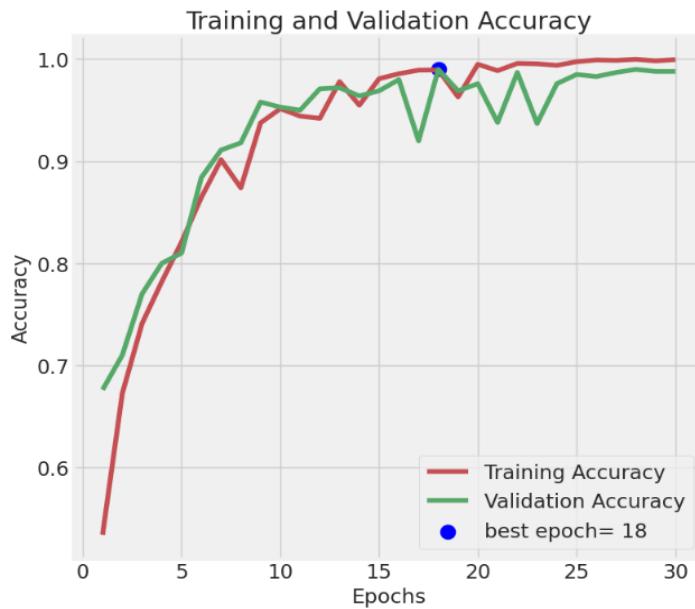
W trakcie trwania procesu treningowego model stopniowo doskonalił swoje umiejętności. Wartość funkcji straty zaczęła systematycznie spadać, a dokładność sukcesywnie rosła w kolejnych epokach. To dowodzi, że model nabywał zdolności do rozpoznawania wzorców w danych.

W kolejnych epokach dokładność modelu dla danych walidacyjnych przekroczyła poziom 90% (już dla 7. epoki), co wskazuje, że model z powodzeniem klasyfikował nowe dane z wysoką dokładnością.

Aby bardziej klarownie przedstawić przebieg treningu, na rys. 4.6 i 4.7 prezentujemy jego wyniki w formie wykresów.



Rysunek 4.6: Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki jelita.



Rysunek 4.7: Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki jelita.

Na rys. 4.6 przedstawiamy wartości funkcji straty na wykresie. Wykres zaczynamy od 2. epoki z uwagi na duży spadek wartości funkcji straty dla zbioru treningowego z 1. epoki na 2. (odczytane z tabeli 4.1: 7,3319 na 0,5950). Widzimy, że wartość funkcji straty dla obu zestawów danych do 15. epoki w większości mała. Świadczy to o poprawności konfiguracji modelu, tzn. o odpowiednim wyborze algorytmu optymalizacyjnego oraz funkcji straty. Od 15. epoki pojawiły się pewne fluktuacje w wynikach dla zbioru walidacyjnego, które od 25. epoki uległy stabilizacji. Ponadto na rys. 4.6 zaznaczono epokę, w której wartość funkcji straty dla zestawu walidacyjnego była najmniejsza.

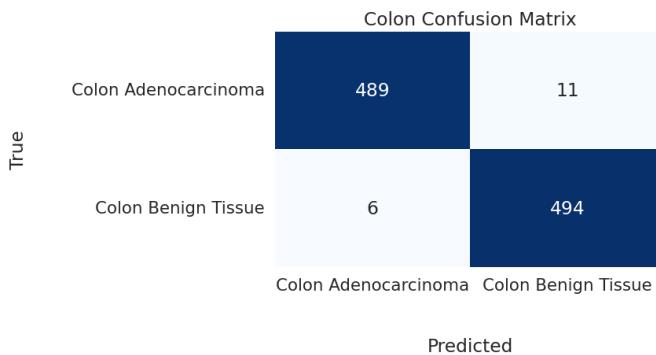
Na rys. 4.7 widzimy, że wartość metryki dokładności dla zestawu treningowego w większości wzrastała. Możemy zauważać, że wartości dokładności dla obu zbiorów w większości były sobie bliskie. Świadczy to o dobrej generalizacji modelu, a więc o tym, że model nie jest przetrenowany (zob. podrozdział 1.5). Podobnie jak na rys. 4.6, na wykresie zaznaczono epokę, w której wartość dokładności była największa.

Warto zwrócić uwagę, że w późniejszych fazach treningu obserwujemy ustabilizowanie się

wydajności modelu lub pojawienie się drobnych fluktuacji. Jest to naturalne zjawisko, ponieważ model zbliżył się do swojego najlepszego poziomu działania.

Zauważamy, że model stosunkowo szybko osiągnął bardzo dobrą wydajność, co sugeruje, że trening mógłby zostać przerwany już w 18 epoce. Niemniej jednak, aby wyrazić fakt, że dokładność modelu stabilizuje się po pewnym czasie i występują jedynie pewne fluktuacje, postanowiliśmy kontynuować trening. Jedną z najpopularniejszych technik przerywania treningu jest technika wczesnego zatrzymania (ang. *early stopping*), wspomniana w podrozdziale 2.6. W treningach naszych sieci pozostaniemy przy konwencji wykonywania wszystkich zaplanowanych epok, w celu obserwacji całosciowej efektywności treningu modelu.

W celu dokładniejszej analizy wydajności naszego modelu, wykonamy predykcję przypadków testowych i porównamy ją z rzeczywistymi danymi. Wykorzystamy do tego celu macierz pomyłek (ang. *confusion matrix*). Na rys. 4.8 przedstawiono macierz pomyłek modelu 1 dla obrazów tkanki jelita.



Rysunek 4.8: Macierz pomyłek modelu 1 dla obrazów tkanki jelita.

Dokładność modelu na zbiorze testowym wyniosła 98,3%, a więc podobnie, jak dla zbioru treningowego. Świadczy to o dobrej generalizacji modelu na nowe przypadki i braku przetrenowania.

Na listingu 4.1 przedstawiamy raport klasyfikacji. Wartości z raportu można wyznaczyć na podstawie macierzy pomyłek.

Classification Report:					
	precision	recall	f1-score	support	
4 Colon Adenocarcinoma	0.9879	0.9780	0.9829	500	
5 Colon Benign Tissue	0.9782	0.9880	0.9831	500	
6					
7 accuracy			0.9830	1000	
8 macro avg	0.9830	0.9830	0.9830	1000	
9 weighted avg	0.9830	0.9830	0.9830	1000	

Listing 4.1: Raport klasyfikacji modelu 1 dla obrazów tkanki jelita.

Na podstawie macierzy pomyłek i raportu klasyfikacji omówimy wartości poszczególnych metryk. Dodajmy, że w raporcie klasyfikacyjnym wartość „macro avg” to średnia z wyników metryk dla poszczególnych klas i jest obliczana, biorąc wyniki metryk z poszczególnych klas i obliczając ich średnią arytmetyczną. Wartość „weighted avg” to średnia ważona wyników metryk, gdzie wagi są przypisane na podstawie liczby próbek w każdej klasie. W naszych przypadkach liczba próbek w klasach jest równa, zatem obie wartości są takie same.

Omówimy pokrótko macierz pomyłek (ang. *confusion matrix*) i metryki z niej wynikające. W macierzy pomyłek każdy rząd reprezentuje klasę rzeczywistą (prawdziwą), a kolumna klasę przewidywaną oznacza to, że liczba wierszy i kolumn odpowiada liczbie klas w zadaniu klasyfikacji. Niekiedy ułożenie etykiet może być odwrotne, tzn. w rzędach predykcje, a w kolumnach etykiety rzeczywiste. Liczba klas może być dowolna, my omówimy macierz dla klasyfikacji binarnej, którą w naturalny sposób można uogólnić do klasyfikacji wieloklasowej. W macierzy pomyłek należy wybrać, która klasa jest pozytywna, a która negatywna. Wybór ten nie ma większego znaczenia, ważne jest rozróżnienie. Na przykład na rys. 4.8 przyjmijmy, że klasa pozytywna to *Colon Adenocarcinoma*, a negatywna *Colon Benign Tissue*. W takim przypadku możemy elementy macierzy w następujący sposób:

- Przypadki prawdziwie pozytywne (ang. *True Positives*, TP): Liczba przypadków, w których model poprawnie przewidział klasę pozytywną (dla danej klasy). Na rys. 4.8: $TP = 489$.
- Przypadki fałszywie pozytywne (ang. *False Positive*, FP): Liczba przypadków, w których model błędnie przewidział klasę pozytywną (dla danej klasy), gdy była to inna klasa. Na rys. 4.8: $TP = 11$.
- Przypadki prawdziwie negatywne (ang. *True Negative*, TN): Liczba przypadków, w których model poprawnie przewidział klasę negatywną (dla danej klasy). Na rys. 4.8: $TN = 494$.
- Przypadki fałszywie negatywne (ang. *False Negative*, FN): Liczba przypadków, w których model błędnie przewidział klasę negatywną (dla danej klasy), gdy była to inna klasa. Na rys. 4.8: $FN = 6$.

Na podstawie tych liczb możemy zdefiniować następujące metryki zadania klasyfikacji:

- Precyza (ang. *precision*): $\frac{TP}{TP+FP}$. Precyza mierzy, ile spośród wszystkich przypadków, które model zaklasyfikował jako pozytywne, naprawdę jest pozytywnych.
- Czułość (ang. *recall*): $\frac{TP}{TP+FN}$. Czułość mierzy, jak wiele z prawdziwie pozytywnych przypadków TP udało się wykryć przez model w stosunku do ogólnej liczby prawdziwie pozytywnych przypadków, czyli $TP + FN$.
- Współczynnik F1 (ang. *F1-score*): $\frac{2 \cdot \text{Precyza} \cdot \text{Czułość}}{\text{Precyza} + \text{Czułość}}$. Wskaźnik F1 to średnia harmoniczna precyzji i czułości, która nadaje większą wagę małym wartościom, w przeciwieństwie np. do średniej arytmetycznej, która traktuje wszystkie wartości jednakowo. Oznacza to, że wysoką wartość wskaźnika F1 otrzymamy, gdy zarówno precyza oraz czułość będą miały wysokie wartości.

Dla klasy nowotworu precyza modelu osiągnęła poziom 98,79%, natomiast czułość 97,80%. Dla klasy tkanki zdrowej precyza modelu osiągnęła poziom 97,82%, a czułość 98,80%. Te wartości świadczą o dobrej jakości naszego modelu, natomiast możemy zauważyc, że model nieco gorzej potrafi rozpoznawać rzeczywiste przypadki klasy tkanki chorej. 11 przypadków klasy tkanki chorej zostało sklasyfikowanych jako tkanki zdrowe, a 6 przypadków klasy tkanki zdrowej zostały sklasyfikowane jako tkanki chore. Z punktu widzenia medycznego ta proporcja, a więc zachowanie modelu, jest mniej zadowalająca niż sytuacja odwrotna, dlatego że te 11 przypadków stanowią realne zagrożenie życia, w przeciwieństwie do 6 przypadków oznaczonych jako tkanki chore będąc rzeczywiście zdrowymi. Mimo wszystko są to bardzo niewielkie liczby pomyłek w stosunku do całości.

Dodatkowo wysoka wartość współczynnika F1 wskazuje na dobrą zdolność predykcyjną modelu. Wartość nośnika (ang. *support*) mówi, że reprezentacja klas w danych testowych była zrównoważona, tj. 500:500.

Model 2 dla obrazów tkanki jelita

Przechodzimy do modelu 2 dla obrazów tkanki jelita.

Przebieg treningu modelu 2, reprezentowanego przez (4.2), przedstawiono w tabeli 4.2.

Tabela 4.2: Przebieg treningu dla modelu 2 dla obrazów tkanki jelita. Przez (T) oznaczono wartości dla danych treningowych, a przez (W) dla danych walidacyjnych.

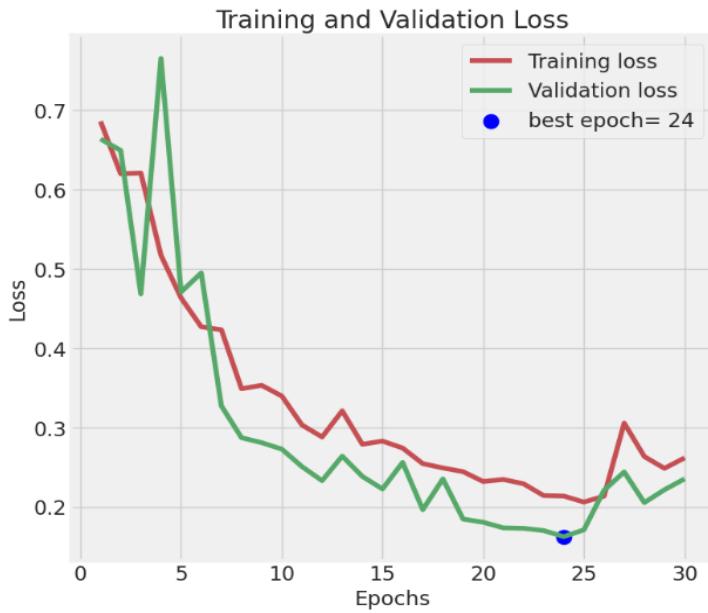
Epoka	Funkcja straty (T)	Dokładność (T)	Funkcja straty (W)	Dokładność (W)
1	0,6859	0,5559	0,6637	0,5000
2	0,6198	0,6615	0,6493	0,6400
3	0,6208	0,6539	0,4684	0,8220
4	0,5171	0,7496	0,7652	0,5620
5	0,4633	0,7854	0,4709	0,7700
6	0,4270	0,8087	0,4948	0,7240
7	0,4232	0,8110	0,3272	0,8590
8	0,3489	0,8549	0,2871	0,8870
9	0,3532	0,8537	0,2809	0,8860
10	0,3397	0,8587	0,2728	0,8950
11	0,3033	0,8769	0,2508	0,9030
12	0,2880	0,8851	0,2330	0,8990
13	0,3210	0,8669	0,2639	0,8930
14	0,2789	0,8871	0,2383	0,9000
15	0,2829	0,8849	0,2225	0,9080
16	0,2741	0,8914	0,2560	0,8920
17	0,2544	0,8984	0,1965	0,9160
18	0,2490	0,9018	0,2352	0,8980
19	0,2444	0,9047	0,1846	0,9310
20	0,2319	0,9110	0,1806	0,9290
21	0,2345	0,9079	0,1735	0,9380
22	0,2291	0,9126	0,1729	0,9340
23	0,2142	0,9161	0,1701	0,9380
24	0,2136	0,9179	0,1625	0,9410
25	0,2060	0,9206	0,1710	0,9320
26	0,2135	0,9170	0,2209	0,9070
27	0,3055	0,8661	0,2440	0,8940
28	0,2634	0,8910	0,2053	0,9230
29	0,2485	0,9031	0,2215	0,9140
30	0,2617	0,8895	0,2353	0,9040

Dokładność na zbiorze treningowym do 6. epoki była poniżej 80%. To sygnalizuje, że model napotykał pewne trudności w dokładnym identyfikowaniu obiektów.

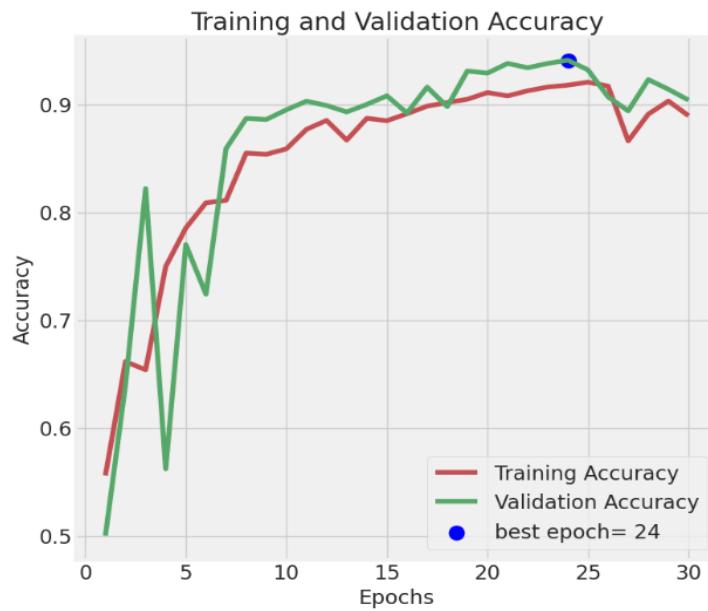
Na rys. 4.9 i 4.10 prezentujemy wyniki przebiegu treningu w formie wykresów.

Możemy zaobserwować, że do 7. epoki wartości dla zestawu walidacyjnego ulegały dużym fluktuacjom, co świadczy o tym, że model nie miał zdolności generalizowania. Natomiast od 7. epoki widzimy, że poza epoką 26. wartości dla zestawu walidacyjnego były lepsze niż dla zestawu treningowego. Jest to sygnał, że proces generalizacji przebiegł bardzo sprawnie.

W trakcie kolejnych epok trwania procesu treningowego model stopniowo poprawiał się. Wartość funkcji straty w dalszym ciągu spadała, poza niewielkimi fluktuacjami, a dokładność wzrosła do 24. epoki.



Rysunek 4.9: Wykres wartości funkcji straty dla treningu modelu 2 dla obrazów tkanki jelita.



Rysunek 4.10: Wykres wartości metryki dokładności dla treningu modelu 2 dla obrazów tkanki jelita.

Warto zwrócić uwagę, że w późniejszych fazach treningu obserwujemy niewielki spadek wydajności modelu oraz pojawienie się drobnych fluktuacji. Widzimy też, że gdybyśmy wykorzystali kryterium wcześniego zatrzymania procesu uczenia to uzyskalibyśmy lepszy model. Patrząc na przebieg uczenia, ostatnia wartość dokładności dla zestawu walidacyjnego jest o kilka punktów procentowych mniejsza niż dla wcześniejszych epok, a zwłaszcza dla epoki 24., która okazała się tą z najlepszymi wartościami parametrów. Widzimy zatem, jak ważne jest wykorzystywanie tej techniki nie tylko, w celu uniknięcia kontynuowania treningu bez progresu, ale i uniknięcia sytuacji spadku wydajności modelu.

Wykonujemy predykcję przypadków testowych i porównujemy ją z rzeczywistymi danymi. Na rys. 4.11 przedstawiono macierz pomyłek modelu 2 dla obrazów tkanki jelita.

		Colon Confusion Matrix	
		Colon Adenocarcinoma	Colon Benign Tissue
True	Colon Adenocarcinoma	477	23
	Colon Benign Tissue	73	427
		Colon Adenocarcinoma	Colon Benign Tissue
		Predicted	

Rysunek 4.11: Macierz pomyłek modelu 2 dla obrazów tkanki jelita.

Dokładność modelu na zbiorze testowym wyniosła 90,4%, a więc wartość podobną, jak dla zbioru walidacyjnego i treningowego.

Na listingu 4.2 przedstawiamy raport klasyfikacji bazujący na macierzy pomyłek.

```

1 Classification Report:
2             precision    recall   f1-score   support
3
4   Colon Adenocarcinoma      0.8673     0.9540     0.9086      500
5   Colon Benign Tissue       0.9489     0.8540     0.8989      500
6
7             accuracy          0.9040
8             macro avg       0.9081     0.9040     0.9038      1000
9             weighted avg    0.9081     0.9040     0.9038      1000

```

Listing 4.2: Raport klasyfikacji modelu 2 dla obrazów tkanki jelita.

Dla klasy obrazów tkanki chorej precyzja modelu osiągnęła poziom 86,73%, natomiast czułość 95,4%. Dla klasy obrazów tkanki zdrowej metryka precyzji wyniosła 94,89%, a czułość 85,4 %. W przypadku tego modelu mamy odwrotną sytuację niż dla modelu 1. Model 2 ma większe problemy z rozpoznaniem klasy bez nowotworu. 73 przypadki klasy obrazów tkanki zdrowej zostały uznane za obrazy tkanki chorej, a 23 przypadki klasy obrazów tkanki chorej model uznał za obrazy tkanki zdrowej. Pomijając skalę tych pomyłek, to punktu widzenia medycznego ta sytuacja jest bardziej korzystna dla modelu diagnostycznego, gdyż stanowi mniejsze ryzyko, że nie rozpoznamy pacjenta, który jest chory.

Model 1 dla obrazów tkanki płucnej

Przechodzimy do modeli dla obrazów tkanki płucnej.

Przebieg treningu modelu 1 dla obrazów tkanki płucnej, zaprezentowanego w podrozdziale 4.6, przedstawiono w tabeli 4.3.

Tabela 4.3: Przebieg treningu modelu 1 dla obrazów tkanki płucnej. Przez (T) oznaczono wartości dla danych treningowych, a przez (W) dla danych walidacyjnych.

Epoka	Funkcja straty (T)	Dokładność (T)	Funkcja straty (W)	Dokładność (W)
1	18,6322	0,7292	0,3430	0,8713
2	0,3398	0,8676	0,3071	0,8733
3	0,2862	0,8905	0,2704	0,8900
4	0,2509	0,9017	0,2047	0,9187
5	0,2098	0,9208	0,1509	0,9420
6	0,1902	0,9265	0,1484	0,9393
7	0,1796	0,9294	0,1408	0,9480
8	0,1619	0,9350	0,1306	0,9527
9	0,1213	0,9515	0,1143	0,9560
10	0,1009	0,9600	0,1192	0,9507
11	0,0989	0,9611	0,1643	0,9380
12	0,0880	0,9661	0,0959	0,9633
13	0,0657	0,9754	0,0801	0,9700
14	0,0551	0,9812	0,1082	0,9700
15	0,0460	0,9842	0,0897	0,9700
16	0,0405	0,9860	0,1193	0,9640
17	0,0383	0,9858	0,0938	0,9740
18	0,0223	0,9914	0,0844	0,9760
19	0,0160	0,9948	0,2085	0,9573
20	0,0295	0,9898	0,1105	0,9747
21	0,0128	0,9968	0,1013	0,9760
22	0,0163	0,9944	0,1436	0,9647
23	0,0172	0,9952	0,0989	0,9767
24	0,0136	0,9964	0,1319	0,9700
25	0,0242	0,9922	0,1291	0,9713
26	0,0130	0,9963	0,0959	0,9780
27	0,0162	0,9945	0,1190	0,9673
28	0,0071	0,9977	0,1025	0,9760
29	0,0044	0,9990	0,1284	0,9740
30	0,0381	0,9874	0,1264	0,9693

Obserwujemy, że generalizacja w tym procesie treningu przebiegała bardzo szybko i skutecznie. Widzimy, że już w 4. epoce wartość dokładności przekroczyła 90% dla obu zbiorów i już poniżej tego progu nie zeszła.

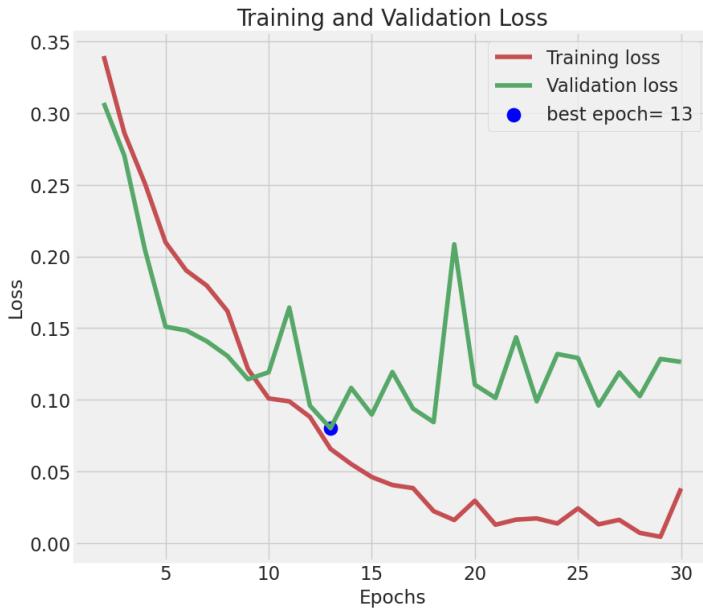
Widzimy, że wartość funkcji straty w 1. epoce dla zbioru treningowego spadła z 18,63322 na 0,3398, stąd na wizualizacji wyników ponownie przedstawimy wykres wartości funkcji straty od 2. epoki.

Na rys. 4.12 i 4.13 prezentujemy wyniki w formie wykresów.

Wartość funkcji straty dla zbioru walidacyjnego od 9. epoki była stosunkowo wyższa niż dla zbioru treningowego.

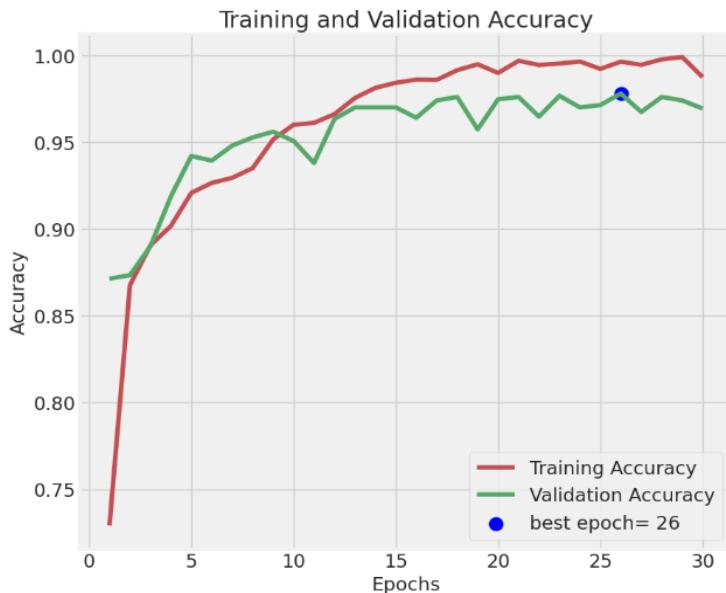
Podobnie w przypadku dokładności, gdzie od 9. epoki wartości dla zbioru walidacyjnego były niższe niż dla zbioru treningowego.

Łatwo zauważyc, że funkcja straty doświadczała fluktuacji, w okolicach wartości od 0,10 do 0,15, ale nie zeszła wiele poniżej 0,10. Natomiast wartość funkcji straty dla zbioru treningowego sukcesywnie spadała (z niewielkimi fluktuacjami).



Rysunek 4.12: Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki płucnej.

Analogicznie w przypadku wartości dokładności. Wartości dla zbioru walidacyjnego „zatrzymał” się ok. 96%, a więc ok. 3 punkty procentowe mniej niż dla zbioru treningowego.



Rysunek 4.13: Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki płucnej.

Poczynione przez nas obserwacje sugerują, że model 2 jest przetrenowany (zob. podrozdział 1.5). Natomiast niekoniecznie jest to powód do obaw, gdyż pewien stopień przetrenowania jest dopuszczalny w przypadku odpowiednio wysokich wyników dla zbioru walidacyjnego.

Na rys. 4.14 prezentujemy macierz pomyłek modelu 1 dla obrazów tkanki płucnej. Dokładność tego modelu na zbiorze testowym wyniosła 96,13%, jest to oznaka bardzo dobrzej generalizacji modelu, pomimo pewnego stopnia przetrenowania.

		Colon Confusion Matrix		
		True	Predicted	
True	Lung Adenocarcinoma	475	2	23
	Lung Benign Tissue	2	498	0
	Lung Squamous Cell Carcinoma	31	0	469
		Lung Adenocarcinoma	Lung Benign Tissue	Lung Squamous Cell Carcinoma

Rysunek 4.14: Macierz pomyłek modelu 1 dla obrazów tkanki płucnej.

Przyjrzymy się dokładniej pozostały metrykom modelu. Na listingu 4.3 umieszczono raport klasyfikacji. Mamy tu do czynienia z klasyfikacją wieloklasową.

1 Classification Report:	precision	recall	f1-score	support
2				
3				
4 Lung Adenocarcinoma	0.9350	0.9500	0.9425	500
5 Lung Benign Tissue	0.9960	0.9960	0.9960	500
6 Lung Squamous Cell Carcinoma	0.9533	0.9380	0.9456	500
7				
8 accuracy			0.9613	1500
9 macro avg	0.9614	0.9613	0.9613	1500
10 weighted avg	0.9614	0.9613	0.9613	1500

Listing 4.3: Raport klasyfikacji modelu 1 dla obrazów tkanki płucnej.

Dla klas gruczolakoraka wartość precyzji wyniosła 93,50%, a więc najmniej w porównaniu do pozostałych klas. Przy 99,60% dla klasy tkanki zdrowej i 95,33% dla klasy raka płaskonablonkowego.

Wartości czułości odpowiednio 93,80% dla klasy raka płaskonablonkowego, 95,00% dla klasy gruczolakoraka i 99,60% dla klasy tkanki zdrowej.

Średnie wartości precyzji, czułości i współczynnika F1 są wszystkie na poziomie ok. 96,13%. W ogólności świadczy to o tym, że nasz model jest skuteczny. Po dokładniejszym przyjrzeniu się wartościom metryk dla poszczególnych klas oraz macierzy pomyłek możemy zauważyc, że nasz model myli się najczęściej (ma największą szansę na pomyłkę) w przypadku klasy odpowiadającej za raka płaskonablonkowego i gruczolakoraka płuca. 23 przypadki gruczolakoraka zostały sklasyfikowane jako rak płaskonablonkowy. Podobna sytuacja ma miejsce w przypadku raka płaskonablonkowego, gdzie 31 przypadków zostało sklasyfikowanych jako gruczolakorak płuca. Dla tkanki wolnej od nowotworu model pomylił się 2 razy. Analizując zdjęcia odpowiadających klas możemy znaleźć prawdopodobne wyjaśnienie tego faktu. Mianowicie, tkanka płuca wolna od nowotworu znacznie różni się od pozostałych klas, co widać na rys. 4.1, 4.2 i 4.3, analizując chociażby kolor zdjęcia. Natomiast obrazy tkanek z nowotworami są do siebie bardziej zbliżone. Może być to wyjaśnienie większej liczby pomyłek dla obrazów klas nowotworowych.

Model 2 dla obrazów tkanki płucnej

Omówimy teraz wyniki dla 2 modelu dla obrazów tkanki płucnej.

Przebieg treningu modelu 2 dla obrazów tkanki płucnej, zaprezentowanego w podrozdziale 4.6, przedstawiono w tabeli 4.4.

Tabela 4.4: Przebieg treningu modelu 2 dla obrazów tkanki płucnej. Przez (T) oznaczono wartości dla danych treningowych, a przez (W) dla danych walidacyjnych.

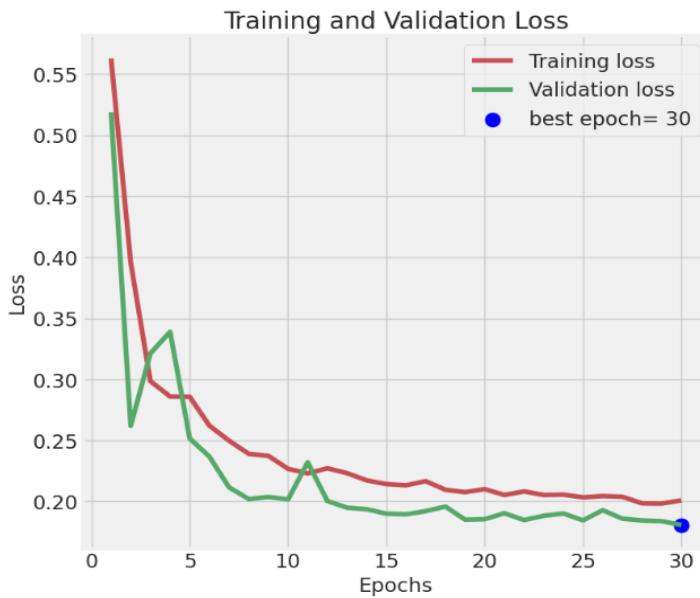
Epoka	Funkcja straty (T)	Dokładność (T)	Funkcja straty (W)	Dokładność (W)
1	0,5627	0,7225	0,5187	0,7580
2	0,3956	0,8290	0,2619	0,8953
3	0,2986	0,8823	0,3213	0,8427
4	0,2860	0,8865	0,3389	0,8580
5	0,2857	0,8869	0,2517	0,8907
6	0,2621	0,8961	0,2368	0,8980
7	0,2498	0,9015	0,2117	0,9133
8	0,2390	0,9078	0,2020	0,9187
9	0,2374	0,9085	0,2038	0,9180
10	0,2268	0,9160	0,2017	0,9153
11	0,2231	0,9116	0,2322	0,8960
12	0,2272	0,9117	0,2004	0,9113
13	0,2232	0,9133	0,1950	0,9233
14	0,2173	0,9164	0,1937	0,9233
15	0,2143	0,9159	0,1901	0,9220
16	0,2133	0,9195	0,1896	0,9173
17	0,2167	0,9154	0,1921	0,9187
18	0,2096	0,9202	0,1959	0,9147
19	0,2077	0,9213	0,1852	0,9253
20	0,2101	0,9206	0,1856	0,9247
21	0,2054	0,9236	0,1904	0,9200
22	0,2083	0,9237	0,1848	0,9240
23	0,2054	0,9206	0,1883	0,9220
24	0,2056	0,9202	0,1902	0,9253
25	0,2035	0,9227	0,1846	0,9247
26	0,2045	0,9217	0,1929	0,9160
27	0,2039	0,9211	0,1860	0,9227
28	0,1985	0,9236	0,1845	0,9260
29	0,1983	0,9231	0,1839	0,9240
30	0,2010	0,9254	0,1808	0,9253

Ponownie dla modelu dla obrazów tkanki płucnej generalizacja w procesie treningu przebiegała bardzo szybko i skutecznie. Od 7. epoki wartość dokładności dla zbioru walidacyjnego była wysoka – powyżej 90%.

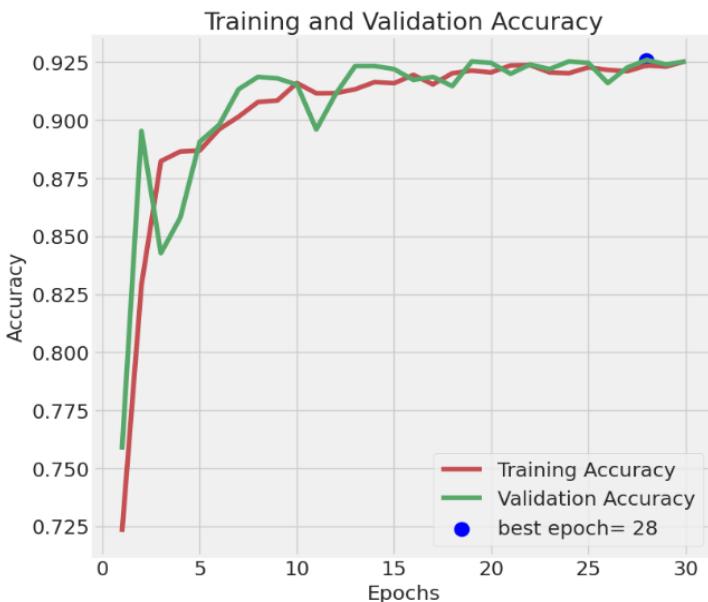
Wartości dokładności i funkcji straty były do siebie bardzo zbliżone dla obu zbiorów. Świadczy to o małym prawdopodobieństwie przetrenowania modelu.

Na rys. 4.15 i 4.16 prezentujemy wyniki w formie wykresów.

Możemy zauważyć, że wartości funkcji straty dla zbioru walidacyjnego były w większości mniejsze niż dla zbioru treningowego oraz w kolejnych epokach malały.



Rysunek 4.15: Wykres wartości funkcji straty dla treningu modelu 2 dla obrazów tkanki płucnej.



Rysunek 4.16: Wykres wartości metryki dokładności dla treningu modelu 2 dla obrazów tkanki płucnej.

Wartości dokładności dla obu zbiorów przez większość epok były zbliżone. Jest to oznaka dobrej generalizacji modelu oraz świadectwo braku przetrenowania.

Dodatkowo możemy zauważać, wartości dokładności, dla obu zbiorów, co prawda wzrastały w kolejnych epokach, ale nie były to znaczne przyrosty.

Po zaznaczonych epokach z najlepszymi wartościami dla dokładności i funkcji straty na rys. 4.15 i rys. 4.16 możemy zauważać, że znajdują się one na końcu procesu treningu. Można wyizzare wniosek, że dobrą decyzją było nieprzerywanie treningu, jednakże wartość ta jest bardzo zbliżona do tych np. z epoki 20. i bez dużych kosztów dokładności moglibyśmy zakończyć wtedy trening z niewiele gorszym modelem.

Na rys. 4.17 prezentujemy macierz pomyłek modelu 2 dla obrazów tkanki płucnej. Dokładność tego modelu na zbiorze testowym wyniosła 92,20%.

Colon Confusion Matrix			
		Predicted	
		Lung Adenocarcinoma	Lung Benign Tissue
True	Lung Adenocarcinoma	441	12
	Lung Benign Tissue	4	496
	Lung Squamous Cell Carcinoma	54	0
			446

Rysunek 4.17: Macierz pomyłek modelu 2 dla obrazów tkanki płucnej.

Przyjrzymy się dokładniej pozostały metrykom modelu. Na listingu 4.4 umieszczono raport klasyfikacji. Oczywiście jest to również przypadek klasyfikacji wieloklasowej.

1	Classification Report:	precision	recall	f1-score	support
2	Lung Adenocarcinoma	0.8838	0.8820	0.8829	500
3	Lung Benign Tissue	0.9764	0.9920	0.9841	500
4	Lung Squamous Cell Carcinoma	0.9047	0.8920	0.8983	500
5				0.9220	1500
6		accuracy			1500
7		macro avg	0.9216	0.9220	1500
8		weighted avg	0.9216	0.9220	1500

Listing 4.4: Raport klasyfikacji modelu 2 dla obrazów tkanki płucnej.

Dla klasy gruczolakoraka precyzaja wyniosła 88,38%, czułość 88,20%. dla klasy raka płaskonabłonkowego precyzaja miała wartość 90,47%, a czułość 89,20%. Natomiast dla klasy tkanki zdrowej odpowiednio 97,64% i 99,20%.

Srednie wartości precyzyji, czułości i współczynnika F1 są wszystkie na poziomie ok. 92,20%. Oznacza to, że nasz model jest skuteczny. Po dokładniejszym przyjrzeniu się wartościom metryk dla poszczególnych klas oraz macierzy pomyłek ponownie obserwujemy, że nasz model myli się najczęściej w przypadku klasy odpowiadającej za raka płaskonabłonkowego i gruczolakoraka. 47 przypadków gruczolakoraka zostało sklasyfikowanych jako rak płaskonabłonkowy. Podobna sytuacja ma miejsce w przypadku raka płaskonabłonkowego, gdzie 54 przypadki zostały sklasyfikowane jako gruczolakorak płuca. Dla tkanki wolnej od nowotworu model pomylił się 4 razy, które to przypadki model uznał za gruczolakoraka. Możemy zatem domyślać się, że nasze przypuszczenia o różnicę w zdjęciach tkanek zdrowych i chorych są prawdziwe i najprawdopodobniej jest to przyczyna lepszej efektywności modeli w przypadkach klasy tkanki zdrowej.

Podsumowanie

Przejdziemy teraz do podsumowania modeli. W tabeli 4.5 przedstawiamy zbiorcze, uśrednione wyniki metryk dla wszystkich modeli.

Tabela 4.5: Podsumowanie stworzonych modeli. Przez (J) oznaczono modele dla obrazów tkanki jelita, a przez (P) dla obrazów tkanki płucnej. Wartości precyzji, czułości i współczynnika F1 są uśrednione.

Model	Metryki			
	Dokładność	Precyzja	Czułość	F1
Model 1 (J)	0,9830	0,9830	0,9830	0,9830
Model 2 (J)	0,9040	0,9081	0,9040	0,9038
Model 1 (P)	0,9613	0,9614	0,9613	0,9613
Model 2 (P)	0,9220	0,9216	0,9220	0,9218

Latwo możemy zauważyć, że w ogólności dużo lepiej poradził sobie model o architekturze 1. Warto jednak podkreślić, że modele o architekturze 1 cechowały się większym przetrenowaniem niż modele o architekturze 2. Widzimy więc, że mniejsza liczba warstw w modelu (tak jak w modelu 1) nie zawsze oznacza mniejsze przetrenowanie modelu. Z drugiej strony, dodanie kolejnej warstwy splotowej nie zawsze musi zwiększyć dokładność modelu w sposób znaczny. Dodatkowo, dzięki wykonaniu eksperymentu, możemy zauważać, że zastosowanie funkcji aktywacji ReLu zwiększa efektywność modelu w porównaniu do zastosowania funkcji aktywacji tangensa hiperbolicznego w sposób znaczący.

Podsumowując, jeśli mielibyśmy wybrać model do wdrożenia w zastosowaniach klinicznych, biorąc pod uwagę powyższe wnioski, to bardziej skłaniałibyśmy się ku wyborowi modelu 1 do tego celu. Oczywiście, nasze rozważania stanowią jedynie wstępne kroki w tak obszernym zadaniu, jakim jest wdrożenie modelu w praktyce klinicznej. Jednak mogą dostarczyć pewnych wskazówek odnośnie tego, w jakim kierunku powinny rozwijać się nasze modele.Więcej na temat możliwości rozwoju naszego projektu omówimy w podrozdziale 4.11.

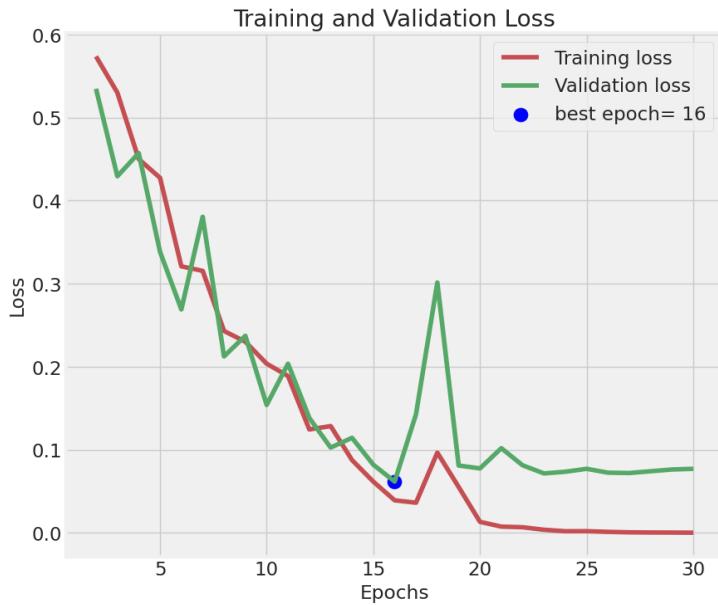
4.8 Eksperimentalne sprawdzenie użyteczności warstwy porzucenia

W celach eksperimentalnych sprawdzamy przebieg treningu modelu 1 dla obrazów tkanki jelita, który pozbawiony jest warstw porzucenia. Wykonując to ćwiczenie chcemy sprawdzić rzeczywistą przydatność wykorzystywania tej warstwy jako techniki regularyzacji.

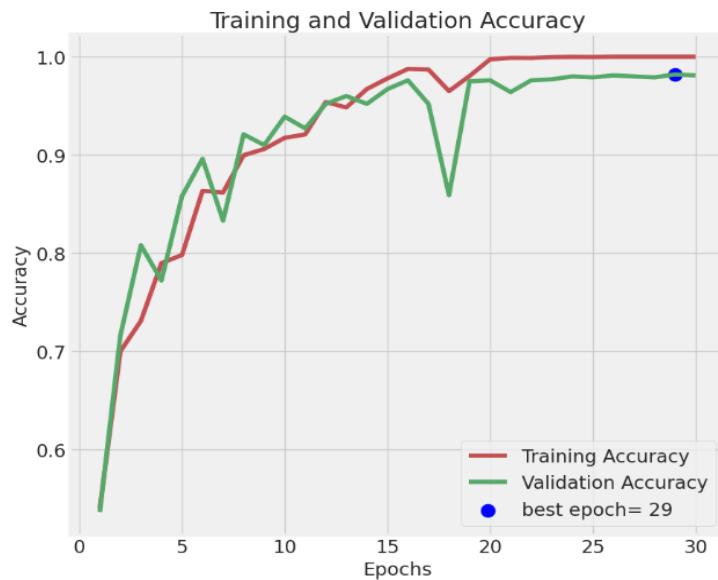
Na rys. 4.18 wykreślamy wartości funkcji straty od 2. epoki z uwagi na duży przeskok. Porównując wykres z wykresem na rys. 4.6 możemy zauważać, że wartość funkcji straty dla modelu bez warstw porzucenia jest większa.

Na rys. 4.19 przedstawiamy wykres wartości dokładności. Porównując go z wykresem na rys. 4.7 widzimy, że przebieg treningu jest z grubsza podobny, natomiast końcowe wartości dokładności na zbiorze walidacyjnym są mniejsze niż dla modelu 1 z rys. 4.7.

Wszystko to oznacza, że model bez warstw porzucenia jest bardziej przetrenowany niż ten, który te warstwy posiada. Możemy zatem potwierdzić użyteczność i sens wykorzystania warstw porzucenia w naszych modelach.



Rysunek 4.18: Wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki jelita bez warstw porzucenia.



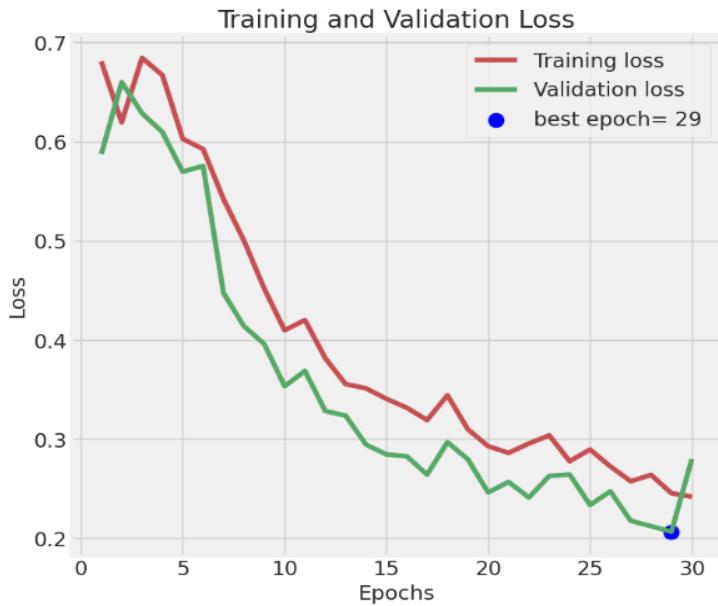
Rysunek 4.19: Wykres wartości metryki dokładności dla treningu modelu 1 dla obrazów tkanki jelita bez warstw porzucenia.

4.9 Eksperyment z inną funkcją aktywacji

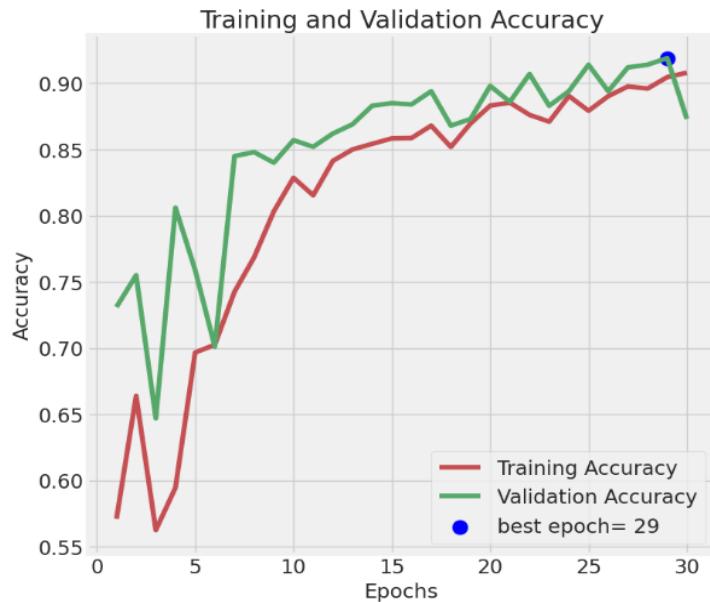
Jako kolejny eksperyment sprawdzimy wydajność modelu 1 dla obrazów tkanki jelita przy zastosowaniu funkcji aktywacji tangensa hiperbolicznego dla warstw splotowych, w sposób analogiczny, jak w modelu 2 dla obrazów tkanki jelita.

Na rys. 4.20 przedstawiamy wykres wartości funkcji straty dla treningu modelu 1 dla obrazów tkanki jelita z funkcją aktywacji tangensa hiperbolicznego. Porównując ten wykres z wykresem na rys. 4.6 możemy zaobserwować, że w ogólności wartości funkcji straty dla modelu z tangensem hiperbolicznym były większe.

Na rys. 4.21 przedstawiamy wykres wartości dokładności dla treningu modelu 1 dla obrazów tkanki jelita z funkcją aktywacji tangensa hiperbolicznego. Porównując ten wykres z wykresem na rys. 4.7. Widzimy, że wartości dokładności dla modelu z tangensem hiperbolicznym są niższe niż dla modelu z funkcją ReLu.



Rysunek 4.20: Wykres wartości funkcji straty dla treningu modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita.



Rysunek 4.21: Wykres wartości metryki dokładności dla treningu modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita.

Na rys. 4.22 przedstawiamy macierz pomyłek dla modelu 1 z funkcją aktywacji tangensa hiperbolicznego na podstawie zbioru testowego. Wartość dokładności tego modelu wynosi 86,70%. Natomiast bez trudu zauważamy, że model dużo gorzej potrafi rozpoznawać przypadki tkanki chorej. Aż 126 chorych tkanek zostało sklasyfikowanych jako tkanki zdrowe. Przyjmując medyczny punkt widzenia ta liczba jest bardzo niepokojąca i w rzeczywistym zastosowaniu mogłaby skutkować dużym zagrożeniem dla zdrowia. Dodatkowo porównując wyniki z modelem 1 dla obrazów tkanki jelita widzimy, że model z funkcją aktywacji tangensa hiperbolicznego ma dużo mniejszą efektywność. Uzyskane wyniki uzasadniają konieczność większego rozbudowania modelu 2 w celu maksymalizacji jego wydajności.

		Colon Confusion Matrix	
		Colon Adenocarcinoma	Colon Benign Tissue
True	Colon Adenocarcinoma	374	126
	Colon Benign Tissue	7	493
		Colon Adenocarcinoma	Colon Benign Tissue
		Predicted	

Rysunek 4.22: Macierz pomyłek modelu 1 z funkcją aktywacji tangensa hiperbolicznego dla obrazów tkanki jelita.

4.10 Wdrożenie modeli

Ze względu na eksperymentalny charakter naszych badań oraz brak współpracy z ośrodkami klinicznymi, nie będziemy w stanie w pełni przeprowadzić tego kroku w ramach procesu CRISP-DM.

W naszym przypadku jako wdrożenie modelu traktujemy zestawienie uzyskanych wyników w postaci raportu oraz umieszczenie ich w niniejszej pracy magisterskiej.

Podjęcie tej kwestii jest istotne, ponieważ podkreśla, że wdrożenie modeli stanowi ważny element procesu CRISP-DM, który polega na praktycznym zastosowaniu stworzonych modeli w rzeczywistym środowisku biznesowym lub naukowym. Celem jest wykorzystanie korzyści płynących z analizy danych w praktyce.

Chociaż brak wdrożenia w placówce klinicznej modeli może uniemożliwić praktyczne wykorzystanie naszych wyników, to nadal istnieje możliwość dalszego badania i udoskonalania opracowanych modeli w przyszłości, gdybyśmy mieli dostęp do odpowiednich danych i środowiska do wdrożenia. Zdobyta wiedza i umiejętności oraz wypracowane wstępne modele mogą być użyteczne w przypadku kontynuacji pracy w przyjętym kierunku, np. we współpracy kliniką lub z Gdańskim Uniwersytetem Medycznym.

4.11 Dalsze możliwości rozwoju

W tym podrozdziale przedstawimy możliwe sposoby na dalszy rozwój naszego projektu.

Pierwszą strategię, którą możemy rozważyć może być rozszerzenie naszego zbioru danych. Pozyskanie większej liczby danych może istotnie wpłynąć na poprawę jakości modelu. Obejmuje to zarówno zwiększenie liczby oryginalnych obrazów pochodzących z zestawów klinicznych, jak i dodatkową augmentację danych. Ogólnie rzecz biorąc, większa liczba obserwacji treningowych przekłada się na lepszą wydajność modelu.

Inną interesującą alternatywą jest wykorzystanie uczenia transferowego, co oznacza korzystanie z istniejących i przeszkołonych już modeli na zbliżonych zadaniach. Uczenie transferowe oznacza możliwość wykorzystania modeli, które już posiadają dużą wiedzę na temat pewnych cech danych, i dostosowania ich do naszych konkretnych potrzeb. To podejście pozwala zaoszczędzić czas i zasoby, które byłyby wymagane do treningu modelu od podstaw, a także może przynieść dobre wyniki, szczególnie w sytuacjach, gdy mamy ograniczone zasoby danych treningowych. Proces polega na dostosowaniu modelu do naszych celów, trenując go na naszym zbiorze danych, a następnie dostrojeniu go w celu uzyskania optymalnej wydajności w konkretnym zadaniu. Ta strategia szczególnie cenna w dziedzinach, gdzie dostępność dużych zbiorów danych jest ograniczona. W tym kontekście warto rozważyć popularne modele, które mogą być przydatne i na przestrzeni lat zdobyły dużą popularność, takie jak LeNet-5, AlexNet, GoogLeNet, lub wspomniany wcześniej model ResNet. Opisy wspomnianych modeli można znaleźć w [7, rozdz. 14], a także w dokumentacji biblioteki Keras [12].

Kolejną opcją jest przetestowanie modeli w praktyce klinicznej poprzez współpracę z ekspertami w dziedzinie onkologii i patomorfologii. To podejście pozwoliłoby nam ocenić, jak te modele sprawdzają się w praktyce, zwłaszcza w realnych sytuacjach klinicznych. Współpraca z ekspertami w dziedzinie medycyny umożliwiłaby zbieranie cennych opinii i wskazówek, które stałyby się wartościowym źródłem informacji do doskonalenia modeli oraz ich dostosowywania do konkretnych potrzeb i wyzwań stawianych przez medycynę.

Inną możliwością jest przeprowadzenie większej liczby eksperymentów sprawdzających kolejne modyfikacje sieci. Na przykład naturalnym rozwinięciem eksperymentu z podrozdziału 4.9 mogłoby być zastosowanie funkcji aktywacji ReLu w modelu 2 i skonfrontowanie efektywności takiego modelu z modelem 1 i modelem 2.

Rozdział 5

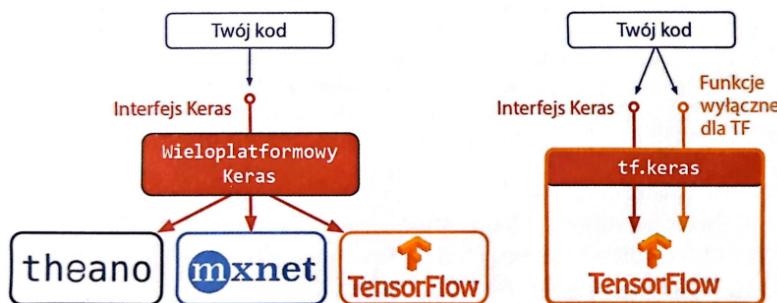
Oprogramowanie

W niniejszym rozdziale omawiamy aspekt programistyczny pracy, prezentując popularne biblioteki w dziedzinie uczenia głębokiego. W celu ułatwienia potencjalnej replikacji naszych wyników udostępniliśmy również kod źródłowy, który został przygotowany specjalnie na potrzeby naszych rozważań.

5.1 Opis bibliotek wykorzystanych w oprogramowaniu

W niniejszym podrozdziale omówimy pokrótko biblioteki, które wykorzystaliśmy w pisaniu kodu programu. W szczególności skupimy się na bibliotekach związanymi z uczeniem głębokim, a więc Keras oraz TensorFlow. Podrozdział powstał w oparciu o dokumentacje bibliotek Keras [12], TensorFlow [29] oraz opis tych bibliotek w [7, rozdz. 10 i rozdz. 12].

Keras jest wysokopoziomowym interfejsem programistycznym (API) przeznaczonym do budowy, trenowania, oceny i wdrażania różnych rodzajów sieci neuronowych w dziedzinie uczenia głębokiego. Ten interfejs jest oparty na różnych platformach do uczenia głębokiego. Obecnie, w najnowszej wersji Keras, możemy korzystać z tego interfejsu z trzema różnymi bibliotekami do uczenia głębokiego: TensorFlow, Microsoft Cognitive Toolkit oraz Theano. Taki wieloplatformowy interfejs Keras pozwala na elastyczne dostosowanie narzędzi do konkretnych potrzeb i preferencji, korzystając z zasobów trzech bibliotek. Na rys. 5.1, w celu lepszego wyobrażenia, przedstawiamy dwie implementacje interfejsu Keras. W omówionej przez nas powyżej implementacji, Keras współpracuje jednocześnie z trzema bibliotekami. Istnieje jednak druga implementacja Keras i jest ona wbudowana w bibliotekę TensorFlow.



Rysunek 5.1: Dwie wersje implementacji interfejsu Keras. Źródło: [7].

Wersja Keras wbudowana w bibliotekę TensorFlow jest jedną z najczęściej wykorzystywanych implementacji. W tej wersji Keras korzysta wyłącznie z funkcji i narzędzi dostępnych w bibliotece TensorFlow, ale dodatkowo oferuje także kilka rozszerzeń i użytecznych funkcji. Z tej implementacji korzystamy również w kodzie programu tej pracy.

Omówimy teraz krótko, czym jest biblioteka TensorFlow. TensorFlow jest przede wszystkim narzędziem do zaawansowanych obliczeń numerycznych, a szczególnie wykorzystuje się go w kontekście skomplikowanego uczenia maszynowego, zwłaszcza w dziedzinie uczenia głębokiego. Początkowo stworzona przez Google, ta biblioteka stała się dostępna publicznie w 2015 roku i od tamtej pory jest liderem w dziedzinie sieci neuronowych w aplikacjach naukowych (zob. [7, rozdz. 12]). Przedstawimy teraz kilka charakterystyk biblioteki TensorFlow, które czynią ją wyjątkowo przydatną.

- Zawiera obsługę procesorów graficznych, co w rezultacie może prowadzić do znacznie szybszych obliczeń w porównaniu do bibliotek, które ich nie obsługują.
- Obsługuje obliczenia rozproszone, czyli takie, które wykorzystują wiele urządzeń i serwerów.
- Implementuje wiele bardzo dobrych algorytmów optymalizacyjnych, takich jak Adamax i stochastyczny gradient prosty.
- Posiada implementację Keras.

5.2 Kod programu

Poniżej umieszczamy kod stworzony na potrzeby pracy. Ponadto kod dostepny jest w repozytorium autora na portalu *GitHub* pod nazwą **MasterThesis_2022-2023**: https://github.com/langkamil/MasterThesis_2022-2023

```
# Print a message to indicate the start of the process
print("Uploading packages...")

# Import necessary libraries
import os                      # For operating system-related functions
import pandas as pd      # For data manipulation and analysis
import numpy as np       # For numerical operations
import tensorflow as tf # For deep learning with TensorFlow
import matplotlib.pyplot as plt # For plotting graphs and charts
import seaborn as sns   # For creating informative and attractive visualizations

# Import specific modules and functions from TensorFlow and scikit-learn
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

# Print a message to indicate that the packages have been successfully loaded
print("Packages loaded.")

def training_validation_plots(history_model, start_epoch=0):
    """
    Function to generate plots of accuracy and loss for training and validation sets.

    Parameters:
    - history_model: Historical training data (output from model training).
    - start_epoch: The epoch number from which to generate the plots (default is 0).

    Output:
    - Two plots: The first one shows training and validation losses,
                  and the second one shows training and validation accuracies.
    """

    # Extract relevant historical training data
    tr_acc = history_model.history['accuracy']           # Training accuracy
    tr_loss = history_model.history['loss']              # Training loss
    val_acc = history_model.history['val_accuracy']      # Validation accuracy
    val_loss = history_model.history['val_loss']         # Validation loss

    # Find the index of the epoch with the lowest validation loss
    index_loss = np.argmin(val_loss)
    val_lowest = val_loss[index_loss]

    # Find the index of the epoch with the highest validation accuracy
    index_acc = np.argmax(val_acc)
    acc_highest = val_acc[index_acc]

    # Create a list of epoch numbers for the x-axis of the plots
    Epochs = [i+1 for i in range(len(tr_acc))]
```

```

# Labels to indicate the best epochs for loss and accuracy
loss_label = f'best epoch= {str(index_loss + 1)}'
acc_label = f'best epoch= {str(index_acc + 1)}'

# Plot training history
sns.set(font_scale=1.5)
plt.figure(figsize=(20, 8), facecolor="w")
plt.style.use('fivethirtyeight')

# Subplot 1: Training and Validation Loss
plt.subplot(1, 2, 1)
plt.plot(Epochs[start_epoch:], tr_loss[start_epoch:], 'r', label= 'Training loss')
plt.plot(Epochs[start_epoch:], val_loss[start_epoch:], 'g',
         label= 'Validation loss')
plt.scatter(index_loss + 1, val_lowest, s= 150, c= 'blue', label= loss_label)
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Subplot 2: Training and Validation Accuracy
plt.subplot(1, 2, 2)
plt.plot(Epochs, tr_acc, 'r', label= 'Training Accuracy')
plt.plot(Epochs, val_acc, 'g', label= 'Validation Accuracy')
plt.scatter(index_acc + 1, acc_highest, s= 150, c= 'blue', label= acc_label)
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Ensure tight layout and display the plots
plt.tight_layout()
plt.show()

def plot_confusion_matrix(test_true_labels, test_predictions, class_names):
    """
    Function to create and display a confusion matrix plot.

    Parameters:
    - test_true_labels: True class labels for the test data.
    - test_predictions: Predicted class labels for the test data.
    - class_names: List of class names for labeling the matrix.

    Output:
    - Displayed confusion matrix plot.
    """

    # Create a confusion matrix
    confusion = confusion_matrix(test_true_labels, test_predictions)

    # Create a figure for the confusion matrix plot
    plt.figure(figsize=(5.7, 3.7), facecolor="w")

    # Set the font scale for better readability
    sns.set(font_scale=1.4)

    # Create a heatmap of the confusion matrix
    sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", cbar=False,
                xticklabels=class_names, yticklabels=class_names)

    # Add labels for the x and y axes
    plt.xlabel("Predicted", labelpad=40)
    plt.ylabel("True", labelpad=40)

```

```

# Add a title to the plot
plt.title("Colon Confusion Matrix")

# Display the plot
plt.show()

# Define the directory where the data is located
data_dir =\
    "/kaggle/input/lung-and-colon-cancer-histopathological-images/lung_colon_image_set"

# Initialize empty lists to store file paths and labels
filepaths = []
labels = []

# List all the subdirectories (folds) in the main data directory
folds = os.listdir(data_dir)

# Iterate through each fold
for fold in folds:
    foldpath = os.path.join(data_dir, fold) # Create the full path to the fold
    flist = os.listdir(foldpath) # List all files in the fold

    # Iterate through each file in the fold
    for f in flist:
        f_path = os.path.join(foldpath, f) # Create the full path to the file
        filelist = os.listdir(f_path) # List all files in the subdirectory

        # Iterate through each file in the subdirectory
        for file in filelist:
            fpath = os.path.join(f_path, file) # Create the full path to the file
            filepaths.append(fpath) # Append the file path to the list

            # Determine the label based on the subdirectory name (fold)
            if f == "colon_aca":
                labels.append("Colon Adenocarcinoma")
            elif f == "colon_n":
                labels.append("Colon Benign Tissue")
            elif f == "lung_aca":
                labels.append("Lung Adenocarcinoma")
            elif f == "lung_n":
                labels.append("Lung Benign Tissue")
            elif f == "lung_scc":
                labels.append("Lung Squamous Cell Carcinoma")

# Create two Pandas Series for file paths and labels
Fseries = pd.Series(filepaths, name="filepaths")
Lseries = pd.Series(labels, name="labels")

# Concatenate the two Series into one DataFrame
df = pd.concat([Fseries, Lseries], axis=1)

# Split colon and lung images into different data frames
df_colon = df[(df["labels"] == "Colon Benign Tissue") |
               (df["labels"] == "Colon Adenocarcinoma")]
df_lung = df[(df["labels"] == "Lung Adenocarcinoma") |
              (df["labels"] == "Lung Benign Tissue") |
              (df["labels"] == "Lung Squamous Cell Carcinoma")]

# Number of lung images for each class
df_lung["labels"].value_counts()

# Number of colon images for each class
df_colon["labels"].value_counts()

```

```

# # Colon

# Split colon images into training, validation, and test subsets

# Extract labels for stratified splitting
strat_colon = df_colon["labels"]

# Split the data into training and temporary subsets with an 80-20 split ratio
train_df_colon, tmp_df_colon = train_test_split(df_colon,
                                                train_size=0.8,
                                                shuffle=True,
                                                random_state=42,
                                                stratify=strat_colon)

# Extract labels for further stratified splitting
strat_colon = tmp_df_colon["labels"]

# Split the temporary subset into validation and test subsets with a 50-50 split ratio
val_df_colon, test_df_colon = train_test_split(tmp_df_colon,
                                               train_size=0.5,
                                               shuffle=True,
                                               random_state=42,
                                               stratify=strat_colon)

# Create generators for train, validation, and test colon data

# Define batch size and image dimensions
batch_size = 128
X = Y = 224

# Create a generator for the training data from the DataFrame train_df_colon
train_generator_colon = ImageDataGenerator().flow_from_dataframe(train_df_colon,
                                                                x_col= "filepaths",           # Column containing file paths
                                                                y_col= "labels",              # Column containing labels
                                                                class_mode = "binary",       # Classification mode
                                                                target_size = (X, Y),        # Target image size
                                                                color_mode="rgb",            # Color mode (RGB)
                                                                batch_size = batch_size,     # Batch size
                                                                shuffle = True,              # Shuffle the data
                                                                seed = 42)                  # Random seed for reproducibility

# Create a generator for the validation data from the DataFrame val_df_colon
val_generator_colon = ImageDataGenerator().flow_from_dataframe(val_df_colon,
                                                                x_col= "filepaths",           # Column containing file paths
                                                                y_col= "labels",              # Column containing labels
                                                                class_mode = "binary",       # Classification mode
                                                                target_size = (X, Y),        # Target image size
                                                                color_mode="rgb",            # Color mode (RGB)
                                                                batch_size = batch_size,     # Batch size
                                                                shuffle = True,              # Shuffle the data
                                                                seed = 42)                  # Random seed for reproducibility

# Create a generator for the test data from the DataFrame test_df_colon
test_generator_colon = ImageDataGenerator().flow_from_dataframe(test_df_colon,
                                                                x_col= "filepaths",           # Column containing file paths
                                                                y_col= "labels",              # Column containing labels
                                                                class_mode = "binary",       # Classification mode
                                                                target_size = (X, Y),        # Target image size
                                                                color_mode="rgb",            # Color mode (RGB)
                                                                batch_size = batch_size,     # Batch size
                                                                shuffle = False,             # Do not shuffle the data
                                                                seed = 42)                  # Random seed for reproducibility

# Get a dictionary mapping class names to their assigned labels

```

```

class_indices = train_generator_colon.class_indices

# Print the dictionary, which shows the mapping of class names to labels
print(class_indices)

# Define a list of class names for colon image classification
class_names_colon = ["Colon Adenocarcinoma", "Colon Benign Tissue"]

# Take true labels from test data generator
test_true_labels_colon = test_generator_colon.classes

# # Colon Model 1

# Create a model architecture for binary classification
model_1_colon = keras.models.Sequential([
    # Convolutional layers with max pooling
    keras.layers.Conv2D(64, 3, activation="relu", padding="same",
                      input_shape=(X, Y, 3)),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(512, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),

    # Flatten the output
    keras.layers.Flatten(),

    # Fully connected layers with dropout for regularization
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dropout(0.5),

    # Output layer for binary classification with sigmoid activation
    keras.layers.Dense(1, activation="sigmoid") # binary classification
])

# Compile the model with an optimizer, loss function, and evaluation metric
model_1_colon.compile(optimizer=tf.keras.optimizers.Adamax(learning_rate=0.001),
                      loss="binary_crossentropy", # binary classification
                      metrics=["accuracy"])

# Display a summary of the model architecture
model_1_colon.summary()

# Train the model
history_1_colon = model_1_colon.fit(train_generator_colon,
                                      epochs=30, # Number of training epochs
                                      validation_data=val_generator_colon,
                                      steps_per_epoch=len(train_generator_colon),
                                      validation_steps=len(val_generator_colon))

#
# Predict on the test data using the trained model
test_predictions_model_1_colon = model_1_colon.predict(test_generator_colon,
                                                       steps=len(test_generator_colon),
                                                       verbose=1)

# Threshold the predicted probabilities to get binary predictions (0 or 1)
test_predictions_model_1_colon = (test_predictions_model_1_colon > 0.5).astype(int)

```

```

# Visualize the training and validation history of model_1_colon
training_validation_plots(history_1_colon)

# Visualize the training and validation history of model_1_colon
training_validation_plots(history_1_colon, start_epoch=1)

# Generate a classification report using the true labels and model predictions
class_report = classification_report(test_true_labels_colon,
                                      test_predictions_model_1_colon,
                                      target_names=class_names_colon,
                                      digits=4)

# Print the classification report
print("Classification Report:")
print(class_report)

# Plot the confusion matrix to visualize model performance
plot_confusion_matrix(test_true_labels=test_generator_colon.classes,
                      test_predictions=test_predictions_model_1_colon,
                      class_names=class_names_colon)

# # Colon Model 2

# Create a model architecture for binary classification (model_2_colon)

# Define the model architecture using Sequential
model_2_colon = keras.models.Sequential([
    # Convolutional layers with max pooling
    keras.layers.Conv2D(64, 3, activation="tanh", padding="same",
                       input_shape=(X, Y, 3)),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(512, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(700, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),

    # Flatten the output
    keras.layers.Flatten(),

    # Fully connected layers with dropout for regularization
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dropout(0.5),

    # Output layer for binary classification with sigmoid activation
    keras.layers.Dense(1, activation="sigmoid") # binary classification
])

# Compile the model with an optimizer, loss function, and evaluation metric
model_2_colon.compile(optimizer=tf.keras.optimizers.legacy.SGD(momentum=0.9,
                                                               learning_rate=0.001,
                                                               decay=0.01),
                      loss="binary_crossentropy", # binary classification
                      metrics=["accuracy"])

# Display a summary of the model architecture
model_2_colon.summary()

```

```

# Train the second model (model_2_colon)

# Fit the model to the training data
history_2_colon = model_2_colon.fit(train_generator_colon,
                                     epochs=30,          # Number of training epochs
                                     validation_data=val_generator_colon,
                                     steps_per_epoch=len(train_generator_colon),
                                     validation_steps=len(val_generator_colon))

# Predict on test data
test_predictions_model_2_colon = model_2_colon.predict(test_generator_colon,
                                                       steps=len(test_generator_colon),
                                                       verbose=1)
test_predictions_model_2_colon = (test_predictions_model_2_colon > 0.5).astype(int)

training_validation_plots(history_2_colon)

class_report = classification_report(test_true_labels_colon,
                                      test_predictions_model_2_colon,
                                      target_names=class_names_colon,
                                      digits=4)
print("Classification Report:")
print(class_report)

plot_confusion_matrix(test_true_labels_colon,
                      test_predictions_model_2_colon,
                      class_names_colon)

# # Lung

# Split lung images into training, validation, and test subsets

# Extract labels for stratified splitting
strat_lung = df_lung["labels"]

# Split the data into training and temporary subsets with an 80-20 split ratio
train_df_lung, tmp_df_lung = train_test_split(df_lung,
                                              train_size=0.8,
                                              shuffle=True,
                                              random_state=42,
                                              stratify=strat_lung)

# Extract labels for further stratified splitting
strat_lung = tmp_df_lung["labels"]

# Split the temporary subset into validation and test subsets with a 50-50 ratio
val_df_lung, test_df_lung = train_test_split(tmp_df_lung,
                                             train_size=0.5,
                                             shuffle=True,
                                             random_state=42,
                                             stratify=strat_lung)

# Define batch size and image dimensions
batch_size = 128
X = Y = 224

```

```

# Create a generator for the training data from the DataFrame train_df_lung
train_generator_lung = ImageDataGenerator().flow_from_dataframe(train_df_lung,
                                                               x_col= "filepaths",           # Column containing file paths
                                                               y_col= "labels",             # Column containing labels
                                                               class_mode = "categorical", # one-hot encoded
                                                               target_size = (X, Y),       # Target image size
                                                               color_mode="rgb",           # Color mode (RGB)
                                                               batch_size = batch_size,    # Batch size
                                                               shuffle = True,             # Shuffle the data
                                                               seed = 42)                  # Random seed for reproducibility

# Create a generator for the validation data from the DataFrame val_df_lung
val_generator_lung = ImageDataGenerator().flow_from_dataframe(val_df_lung,
                                                               x_col= "filepaths",           # Column containing file paths
                                                               y_col= "labels",             # Column containing labels
                                                               class_mode = "categorical", # one-hot encoded
                                                               target_size = (X, Y),       # Target image size
                                                               color_mode="rgb",           # Color mode (RGB)
                                                               batch_size = batch_size,    # Batch size
                                                               shuffle = True,             # Shuffle the data
                                                               seed = 42)                  # Random seed for reproducibility

# Create a generator for the test data from the DataFrame test_df_lung
test_generator_lung = ImageDataGenerator().flow_from_dataframe(test_df_lung,
                                                               x_col= "filepaths",           # Column containing file paths
                                                               y_col= "labels",             # Column containing labels
                                                               class_mode = "categorical", # one-hot encoded
                                                               target_size = (X, Y),       # Target image size
                                                               color_mode="rgb",           # Color mode (RGB)
                                                               batch_size = batch_size,    # Batch size
                                                               shuffle = False,            # Do not shuffle the data
                                                               seed = 42)                  # Random seed for reproducibility

# Get a dictionary mapping class names to their assigned labels
class_indices = train_generator_lung.class_indices

# Print the dictionary, which shows the mapping of class names to labels
print(class_indices)

# Define a list of class names for lung image classification
class_names_lung = ["Lung Adenocarcinoma",
                    "Lung Benign Tissue",
                    "Lung Squamous Cell Carcinoma"]

# Take true labels from test data generator
test_true_labels_lung = test_generator_lung.classes

# # Lung Model 1

# Define the number of classes based on the class indices
class_number = len(list(train_generator_lung.class_indices.keys()))

model_1_lung = keras.models.Sequential([
    # Convolutional layers with max pooling
    keras.layers.Conv2D(64, 3, activation="relu", padding="same",
                       input_shape=(X, Y, 3)),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(512, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),

```

```

# Flatten the output
keras.layers.Flatten(),

# Fully connected layers with dropout for regularization
keras.layers.Dense(512, activation="relu"),
keras.layers.Dropout(0.5),
keras.layers.Dense(256, activation="relu"),
keras.layers.Dropout(0.5),
# Multi-class classification using softmax activation
keras.layers.Dense(class_number, activation="softmax")
])

# Compile model
model_1_lung.compile(optimizer=tf.keras.optimizers.Adamax(learning_rate= 0.001),
                      loss="categorical_crossentropy", # multi-class
                      metrics=["accuracy"])

# Show model summary
model_1_lung.summary()

# Train model
history_1_lung = model_1_lung.fit(train_generator_lung,
                                    epochs=30,
                                    validation_data=val_generator_lung,
                                    steps_per_epoch=len(train_generator_lung),
                                    validation_steps=len(val_generator_lung))

# Predict class probabilities for the test data using the trained model
test_predictions_model_1_lung = model_1_lung.predict(test_generator_lung)

# Extract the class labels with the highest predicted probabilities for each sample
test_predictions_model_1_lung = np.argmax(test_predictions_model_1_lung, axis=1)

# Visualize the training and validation history of model_1_lung
training_validation_plots(history_1_lung)
training_validation_plots(history_1_lung, start_epoch=1)

# Generate a classification report using the true labels and model predictions
class_report = classification_report(test_true_labels_lung,
                                      test_predictions_model_1_lung,
                                      target_names=class_names_lung,
                                      digits=4)

# Print the classification report
print("Classification Report:")
print(class_report)

plot_confusion_matrix(test_true_labels_lung,
                      test_predictions_model_1_lung,
                      class_names_lung)

# # Lung Model 2

class_number = len(list(train_generator_lung.class_indices.keys()))

# Create model architecture for multi-class classification
model_2_lung = keras.models.Sequential([

```

```

        keras.layers.Conv2D(64, 3, activation="tanh", padding="same",
                           input_shape=(X, Y, 3)),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(128, 3, activation="tanh", padding="same"),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(256, 3, activation="tanh", padding="same"),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(512, 3, activation="tanh", padding="same"),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(700, 3, activation="tanh", padding="same"),
        keras.layers.MaxPooling2D(2),
        keras.layers.Flatten(),
        keras.layers.Dense(512, activation="relu"),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(256, activation="relu"),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(class_number, activation="softmax") # multi-class
    ])

# Compile model
model_2_lung.compile(optimizer=tf.keras.optimizers.SGD(momentum=0.9,
                                                       learning_rate=0.001,
                                                       decay=0.01),
                      loss="categorical_crossentropy",
                      metrics=["accuracy"])

# Show model summary
model_2_lung.summary()

# Train model
history_2_lung = model_2_lung.fit(train_generator_lung,
                                    epochs=30,
                                    validation_data=val_generator_lung,
                                    steps_per_epoch=len(train_generator_lung),
                                    validation_steps=len(val_generator_lung))

test_predictions_model_2_lung = model_2_lung.predict(test_generator_lung)
test_predictions_model_2_lung = np.argmax(test_predictions_model_2_lung, axis=1)

training_validation_plots(history_2_lung)

# Predict on test data
test_predictions = model_2_lung.predict(test_generator_lung,
                                         steps=len(test_generator_lung),
                                         verbose=1)

# Take true labels from test data generator
test_true_labels = test_generator_lung.classes

# Generate a classification report using the true labels and model predictions
class_report = classification_report(test_true_labels_lung,
                                       test_predictions_model_2_lung,
                                       target_names=class_names_lung,
                                       digits=4)
print("Classification Report:")
print(class_report)

plot_confusion_matrix(test_true_labels_lung,
                      test_predictions_model_2_lung,
                      class_names_lung)

# # Experiments

```

```

# Create a model architecture for binary classification without dropout

# Define the model architecture using Sequential
model_1_colon_no_dropout = keras.models.Sequential([
    keras.layers.Conv2D(64, 3, activation="relu", padding="same",
                       input_shape=(X, Y, 3)),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(512, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid") # binary classification
])

# Compile the model with an optimizer, loss function, and evaluation metric
model_1_colon_no_dropout.compile(optimizer=tf.keras.optimizers.Adamax(learning_rate=0.001),
                                  loss="binary_crossentropy",
                                  metrics=["accuracy"])

# Display a summary of the model architecture
model_1_colon_no_dropout.summary()

# Train model
history_1_colon_no_dropout = model_1_colon_no_dropout.fit(train_generator_colon,
                                                          epochs=30,
                                                          validation_data=val_generator_colon,
                                                          steps_per_epoch=len(train_generator_colon),
                                                          validation_steps=len(val_generator_colon))

training_validation_plots(history_1_colon_no_dropout)

training_validation_plots(history_1_colon_no_dropout, start_epoch=1)

# Create a model architecture for binary classification with "tanh" activation

# Define the model architecture using Sequential
model_1_colon_tanh = keras.models.Sequential([
    keras.layers.Conv2D(64, 3, activation="tanh", padding="same",
                       input_shape=(X, Y, 3)),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(512, 3, activation="tanh", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1, activation="sigmoid") # binary classification
])

# Compile the model with an optimizer, loss function, and evaluation metric
model_1_colon_tanh.compile(optimizer=tf.keras.optimizers.SGD(momentum=0.9,
                                                               learning_rate=0.001,

```

```

        decay=0.01),
    loss="binary_crossentropy", # Binary cross-entropy loss
    metrics=["accuracy"])

# Display a summary of the model architecture
model_1_colon_tanh.summary()

# Train model
history_1_colon_tanh = model_1_colon_tanh.fit(train_generator_colon,
    epochs=30,
    validation_data=val_generator_colon,
    steps_per_epoch=len(train_generator_colon),
    validation_steps=len(val_generator_colon))

training_validation_plots(history_1_colon_tanh)

# Predict on test data
test_predictions_model_1_colon_tanh = model_1_colon_tanh.predict(test_generator_colon,
    steps=len(test_generator_colon),
    verbose=1)
# binary classification threshold
test_predictions_model_1_colon_tanh =\
    (test_predictions_model_1_colon_tanh > 0.5).astype(int)

# Generate a classification report using the true labels and model predictions
class_report = classification_report(test_true_labels_colon,
    test_predictions_model_1_colon_tanh,
    target_names=class_names_colon,
    digits=4)

# Print the classification report
print("Classification Report:")
print(class_report)

plot_confusion_matrix(test_true_labels_colon,
    test_predictions_model_1_colon_tanh,
    class_names_colon)

```

Podsumowanie

Celem niniejszej pracy magisterskiej było opisanie wybranych metod uczenia głębokiego sieci neuronowych w analizie danych medycznych. W dążeniu do tego celu przedstawiliśmy kluczowe koncepcje z obszaru uczenia maszynowego, uogólnionych modeli liniowych i procesu eksploracyjnej analizy danych, co stanowi niezbędne tło do zrozumienia tematu pracy oraz pozwala na bardziej obszerne spojrzenie na rozległą dziedzinę jaką jest uczenie maszynowe i w konsekwencji uczenie głębokie.

Skoncentrowaliśmy się na zaprezentowaniu podstaw teorii sztucznych sieci neuronowych. Rozpoczęliśmy od przystępnego wprowadzenia do jednokierunkowej sztucznej sieci neuronowej, a także dostarczyliśmy praktyczny przykład w postaci perceptronu i jego rozwinięcia do perceptronu wielowarstwowego. Nasze podejście do tematu było zarówno intuicyjne, aby można było zrozumieć ogólną ideę, jak i matematycznie formalne, co umożliwia zgłębienie bardziej technicznych aspektów teorii. Na koniec opisaliśmy również algorytmy treningu sztucznych sieci neuronowych, co stanowi istotny krok w procesie ich zrozumienia i efektywnego wykorzystania.

Kluczowym punktem części teoretycznej pracy była analiza zastosowań uczenia głębokiego w przetwarzaniu obrazów, ze szczególnym uwzględnieniem sieci splotowych. Przedstawiliśmy elementy architektury takiej sieci w sposób intuicyjny, ułatwiający zrozumienie założeń, oraz formalny, co pozwoliło na dogłębne poznanie ich struktury i mechanizmu. Dodatkowo, ilustrowaliśmy te koncepcje przykładami, aby ułatwić praktyczne zrozumienie przedstawianych zagadnień.

Część eksperymentalną pracy rozpoczęliśmy od wybiórczego przeglądu literatury, który dał nam wgląd we współczesne zastosowania sztucznych sieci neuronowych w medycynie oraz unaocznił mnogość możliwych podejść do zagadnień eksploracji danych z wykorzystaniem metod uczenia głębokiego. W dalszej części pracy przedstawiliśmy wyszukany przez nas duży i ogólnodostępny zbiór danych medycznych wysokiej jakości. Zbudowaliśmy cztery własne architektury sieci splotowych, bazując po części na popularnych praktykach stosowanych wśród specjalistów z dziedziny uczenia maszynowego oraz przykładach literaturowych. Zamysł różnic w architekturach sieci także nie był przypadkowy. Wybraliśmy je z zamiarem skonfrontowania niektórych popularnych praktyk oraz przyjętych standardów w uczeniu głębokim celem sprawdzenia ich zaasadności. W kolejnym etapie badań, przedstawiliśmy wyniki efektywności naszych modeli w sposób indywidualny, który wyleminował wszelkie wątpliwości dotyczące ich zalet oraz ograniczeń. Podsumowaliśmy wydajność architektur sieci, uwzględniając ich różnice, które zostały umotywowane na etapie ich konstrukcji. Ponadto, przeprowadziliśmy dwa eksperymenty, które wnoszą dodatkową wartość poznawczą dla przyszłych konstrukcji architektur, a także ukierunkowują możliwości dalszych badań. Rozwijając temat przyszłych perspektyw badawczych, sprecyzowaliśmy potencjalne kierunki dalszego rozwoju naszej pracy.

Warto zaznaczyć, że kolejne etapy części eksperymentalnej zrealizowaliśmy w duchu koncepcji standardowego procesu eksploracji danych CRISP-DM omówionego w części teoretycznej. Ponadto wykonana analiza odwoływała się do teoretycznych aspektów rozpatrywanych w pracy. W ten sposób zachowaliśmy spójny charakter pracy, co uwydatnia konieczność wnikliwego opisu teorii dla dogłębniego zrozumienia jej zastosowania.

Opis części eksperymentalnej dopełniliśmy omówieniem aspektu programistycznego, prezentując popularne biblioteki w dziedzinie uczenia głębokiego. W celu ułatwienia potencjalnej replikacji naszych wyników udostępniliśmy również kod źródłowy pracy, który został przygotowany specjalnie na jej potrzeby.

Podsumowując naszą pracę, chcemy podkreślić znaczenie oraz ogromny potencjał, jakie niesie ze sobą dziedzina uczenia głębokiego, a także ogólnie mówiąc, sztuczna inteligencja, w kontekście medycyny. Odkrycia i innowacje w tej dziedzinie otwierają nowe perspektywy i możliwości dla przyszłości medycyny, umożliwiając lepsze diagnozowanie, leczenie i opiekę nad pacjentami. Obecny postęp w dziedzinie uczenia maszynowego i sztucznej inteligencji istotnie przyczynia się do znaczącej poprawy jakości opieki zdrowotnej, co ma niebagatelne znaczenie dla ratowania ludzkiego życia i zdrowia. Dalsze badania w tym obszarze mogą przynieść wiele korzyści społeczeństwu, do czego niniejsza praca może stanowić skromy wkład.

Bibliografia

- [1] Bishop C. M., *Neural Networks for Pattern Recognition*, Nowy Jork, Oxford University Press, 1995.
- [2] Bonaccorso G. (tłum. Sawka K.), *Algorytmy uczenia maszynowego. Zaawansowane techniki implementacji*, Helion, 2019.
- [3] Borkowski A. A., Bui M. M., Thomas L. B., Wilson C. P., DeLand L. A., Mastorides S. M., *Lung and Colon Cancer Histopathological Image Dataset (LC25000)*, arXiv:1912.12142v1, 2019.
- [4] Brownlee J., *How Do Convolutional Layers Work in Deep Learning Neural Networks?* <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/> (data dostępu: 15.09.2023).
- [5] Burkov A., *The Hundred-Page Machine Learning Book*, Andriej Burkov, 2019.
- [6] Chan S. H., *Introduction to Probability for Data Science*, Michigan Publishing, 2023.
- [7] Géron A. (tłum. Sawka K.), *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow*, wyd. II, Helion, 2020.
- [8] Glembin M, Obuchowski A., Klaudel B., Rydzinski B., Karski R., Syty P., Jasik P., Narozanski W. J., *Improving Renal Tumor Diagnosis with Computed Tomography and Artificial Neural Networks*, Medical Science Monitor: International Medical Journal of Experimental and Clinical Research, 29:e939462, 2023.
- [9] Goodfellow I., Bengio Y., Courville A., *Deep Learning*, MIT Press, 2016.
- [10] Ivakhnenko A. G., *Cybernetic Predicting Devices*, CCM Information Corporation, 1973.
- [11] Jakubowski J., Sztencel R., *Wstęp do teorii prawdopodobieństwa*, wyd. IV, Warszawa, SCRIFT, 2010.
- [12] Keras API Reference, <https://keras.io/api/> (data dostępu: 15.09.2023).
- [13] Kingma D. P., Ba J., *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980, 2015.
- [14] Koushik J., *Understanding Convolutional Neural Networks*, arXiv:1605.09081v1, 2016.
- [15] Larose D. T. (tłum. Wilbik A.), *Metody i modele eksploracji danych*, Warszawa, PWN, 2012.
- [16] Larose D. T., Larose C. D., *Discovering Knowledge in Data. An Introduction to Data Mining*, wyd. II, Hoboken, Wiley, 2014.
- [17] Larxel, *Lung and Colon Cancer Histopathological Images* <https://www.kaggle.com/datasets/andrewmvd/lung-and-colon-cancer-histopathological-images> (data dostępu: 30.04.2023).
- [18] Liu M., Li L., Wang H., Guo X., Liu Y., Li Y., Song K. Shao Y., Wu F. Zhang J., Sun N., Zhang T., Luan L., *A multilayer perceptron-based model applied to histopathology image classification of lung adenocarcinoma subtypes*, Frontiers in Oncology, 13:1172234, 2023.
- [19] Mitchell T. M., *The Discipline of Machine Learning*, Pittsburgh, Carnegie Mellon University, 2006.
- [20] *Multi-layer Perceptron in TensorFlow*, <https://www.javatpoint.com/multi-layer-perceptron-in-tensorflow> (data dostępu: 12.08.2023).

- [21] Murphy K. P., *Machine Learning: A Probabilistic Perspective*, The MIT Press, 2012.
- [22] ResNet and ResNetV2
<https://keras.io/api/applications/resnet/> (data dostępu: 13.09.2023).
- [23] Rosenblatt F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Waszyngton, Spartan Books, 1962.
- [24] Saha S., *A Comprehensive Guide to Convolutional Neural Networks – the ELI5 way*, <https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/> (data dostępu: 11.08.2023).
- [25] Sidharth, *The Perceptron Algorithm: From Scratch Using Python*
<https://www.pycodemates.com/2022/12/perceptron-algorithm-understanding-and-implementation-python.html> (data dostępu: 26.08.2023).
- [26] Srikanthamurthy M. M., Rallabandi V. P. S., Dudekula D. B., Natarajan S., Park J., *Classification of benign and malignant subtypes of breast cancer histopathology imaging using hybrid CNN-LSTM based transfer learning*, BMC Medical Imaging, 23:19, 2023.
- [27] Szeliga M., *Data science i uczenie maszynowe*, Warszawa, PWN, 2017.
- [28] Szeliga M., *Praktyczne uczenie maszynowe*, Warszawa, PWN, 2019.
- [29] TensorFlow Python API, https://www.tensorflow.org/api_docs/python/tf (data dostępu: 17.09.2023).
- [30] Wirth R., Hipp J., *CRISP-DM: Towards a Standard Process Model for Data Mining*, Journal of Data Warehousing, 10.1.1.198.5133, 1999.
- [31] Wütrich M. V., Merz M., *Statistical Foundations of Actuarial Learning and its Applications*, Springer, 2023.
- [32] Yamashita R., Nishio M., Kinh Gian Do R., Togashi K., *Convolutional neural networks: an overview and application in radiology*, Springer, 2018.