

UNIwersYTET ZIELONOGÓRSKI

Wydział Informatyki, Elektrotechniki i Automatyki

Praca magisterska

Kierunek: Informatyka

**WYKORZYSTANIE GŁĘBOKICH SIECI
NEURONOWYCH DO WSPOMAGANIA
DIAGNOSTYKI MEDYCZNEJ**

inż. Krystian Dziędzioła

Promotor:
dr hab. inż. Artur Gramacki

Zielona Góra, czerwiec 2019

Streszczenie

Głównym celem pracy jest przedstawienie wyników przeprowadzonych badań dotyczących możliwości zastosowania głębokich sieci neuronowych do wspomagania diagnostyki medycznej. Przykładowym zadaniem, które zostało wybrane jako obiekt badań jest diagnozowanie stanów padaczkowych na podstawie odczytów z elektroencefalogramu (EEG).

W pierwszej części pracy zawarte zostały informacje teoretyczne obejmujące zagadnienia sztucznej inteligencji, uczenia maszynowego oraz klasycznych i głębokich sieci neuronowych. Przedstawiony został również wybrany problem praktyczny z dziedziny diagnostyki medycznej.

Druga część przedstawia przegląd narzędzi programistycznych wykorzystywanych do implementacji głębokich sieci neuronowych oraz implementację procesu nauki sieci neuronowej. Opisane zostały również przeprowadzone badania mające na celu sprawdzenie czy głębokie sieci neuronowe można zastosować do rozwiązania wybranego problemu.

Motywacją do podjęcia tematu pracy była chęć rozpoznania popularnego w dzisiejszych czasach zagadnienia głębokich sieci neuronowych oraz próba zastosowania ich dla rzeczywistego problemu praktycznego.

Słowa kluczowe: Sieci neuronowe, diagnostyka medyczna, EEG, deep learning.

Spis treści

1. Wstęp	1
1.1. Wprowadzenie	1
1.2. Cel i zakres pracy	1
1.3. Przegląd literatury	2
1.4. Struktura pracy	2
2. Wprowadzenie do głębokich sieci neuronowych	3
2.1. Deep learning, czyli głębokie sieci neuronowe	3
2.2. Porównanie klasycznych oraz głębokich sieci neuronowych	4
2.3. Architektury głębokich sieci neuronowych	5
2.3.1. Splotowe sieci neuronowe	5
2.3.2. Rekurencyjne sieci neuronowe	6
3. Omówienie wybranego problemu diagnostyki medycznej	8
3.1. Przedstawienie problemu	8
3.2. Dostępne dane	9
3.3. Oczekiwane rezultaty	9
4. Przegląd dostępnych bibliotek oraz narzędzi programistycznych	10
4.1. Dostępne narzędzia	10
4.2. Wybrany stos technologiczny	11
5. Rozwiązanie problemu	12
5.1. Przygotowanie danych	12
5.2. Metoda oceny wyników	22
5.2.1. Hold-out	23
5.2.2. Walidacja krzyżowa	23
5.2.3. Implementacja walidacji	24
5.3. Uruchomienie uczenia właściwego	25
5.3.1. Budowa modelu	26
5.3.2. Lista callback'ów	27
5.3.3. Funkcja dopasowania	28
5.3.4. Sprawdzenie skuteczności	28
5.4. Monitorowanie	29
5.5. Wybór rodzaju sieci neuronowej	32
5.5.1. Klasyczna sieć neuronowa typu fully connected	32
5.5.2. Sieć splotowa z filtrem 1-wymiarowym	33
5.5.3. Sieć splotowa z filtrem 2-wymiarowym	35
5.5.4. Sieć rekurencyjna	36
5.5.5. Połączenie sieci splotowej z rekurencyjną	38

5.5.6. Podsumowanie	39
5.6. Optymalizacja	41
5.6.1. Batch normalization	42
5.6.2. Dropout	42
5.6.3. Wybór optymalizatora	43
5.6.4. Adaptacyjny współczynnik uczenia	44
5.6.5. Modyfikacja rozmiaru sieci	45
5.7. Weryfikacja i ocena otrzymanych rezultatów	48
6. Podsumowanie i wnioski końcowe	51
6.1. Możliwości rozwoju	51
A. Użycie biblioteki Keras w języku R	52
A.1. Implementacja przykładowego modelu do klasyfikacji danych MNIST	52
A.2. Model sieci do rozpoznawania stanów padaczkowych w języku R . . .	54
A.3. Podsumowanie	55
B. Płyta DVD	56

Spis rysunków

2.1. Schemat procesu uczenia głębokiej sieci neuronowej	4
2.2. Porównanie struktury sieci płytkiej oraz głębokiej	4
2.3. Dane liczbowe obrazka w formacie RGB	6
2.4. Schemat działania sieci splotowej. Źródło: [1]	6
2.5. Schemat rekurencyjnej sieci neuronowej	7
5.1. Rozmieszczenie elektrod w systemie 10-20. Źródło: [2]	13
5.2. Wykres przedstawiający odczyt EEG z zaznaczonym momentem wystąpienia ataku	16
5.3. Odczyty ze wszystkich 16 kanałów z zaznaczonymi miejscami wystąpienia ataków	17
5.4. Podział danych na zbiory: treningowy, walidacyjny i testowy	23
5.5. Podział danych według metody k-krotnej walidacji krzyżowe	24
5.6. Dynamicznie tworzone wykresy w programie TensorBoard	30
5.7. Fragment schematu modelu wygenerowany przez TensorBoard	31
5.8. Wyniki skuteczności sieci typu <i>fully-connected</i>	33
5.9. Wyniki prostej sieci splotowej z 1-wymiarowym filtrem	35
5.10. Wyniki prostej sieci splotowej z 2-wymiarowym filtrem	37
5.11. Wyniki prostej sieci rekurencyjnej	38
5.12. Wyniki sieci konwolucyjnej połączonej z rekurencyjną	40
5.13. Wyniki modelu początkowego konwolucyjnej sieci neuronowej	41
5.14. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami <i>Batch-Normalization</i>	43
5.15. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami <i>Dropout</i>	44
5.16. Wyniki modelu po dodaniu zmiennego współczynnika uczenia	45
A.1. Przykładowe dane ze zbioru MNIST	53

Spis tabel

5.1. Porównanie czasów nauki oraz skuteczności poszczególnych typów sieci	39
5.2. Porównanie skuteczności sieci przy użyciu poszczególnych optymalizatorów	44

Rozdział 1

Wstęp

1.1. Wprowadzenie

Sztucznej inteligencja, której dużą część stanowią sztuczne sieci neuronowe, pozwala na wykonywanie przez maszyny zadań, które do niedawna byli w stanie wykonywać tylko ludzie. Jednym z przykładów jest diagnostyka medyczna, która wymaga specjalistycznej wiedzy oraz doświadczenia.

Dzięki sztucznym sieciom neuronowym maszyna jest w stanie nauczyć się pewnych reguł, na podstawie których podejmuje decyzje, które mogą symulować ludzką inteligencję. W niektórych problemach maszyna jest w stanie podejmować poprawne decyzje z dokładnością dorównującą lub nawet przewyższającą zdolności człowieka. Dodatkowo może to zrobić w znacznie krótszym czasie.

Kolejnym wcieleniem klasycznych sieci neuronowych są głębokie sieci neuronowe (ang. *deep neural networks*), które znacząco zwiększają ich możliwości oraz obszar zastosowań. Ewolucja sieci neuronowych była możliwa dzięki szybkiemu rozwojowi sprzętu komputerowego, który oferuje znacząco większe moce obliczeniowe, co pozwala na efektywne uczenie bardziej skomplikowanych struktur sieci. Nowe możliwości sieci neuronowych spowodowały znaczący wzrost ich populaności.

W tekście pracy przedstawiony został opis badań mających na celu sprawdzenie możliwości zastosowania głębokich sieci neuronowych do diagnostyki napadów padaczkowych na podstawie odczytów z elektroencefalogramu (EEG).

1.2. Cel i zakres pracy

Celem pracy jest próba wykorzystania głębokich sieci neuronowych do wspomagania diagnostyki medycznej dla wybranego problemu praktycznego oraz osiągnięcie możliwie najlepszych rezultatów.

Zakres pracy obejmuje:

- krótkie wprowadzenie w tematykę sieci głębokich oraz porównanie z klasycznymi sieciami neuronowymi,
- przegląd dostępnych bibliotek i narzędzi programistycznych dla sieci głębokich,

- omówienie praktycznego problemu z dziedziny diagnostyki medycznej i próba rozwiązania go z wykorzystaniem sieci głębokich,
- przedstawienie szczegółów technicznych i implementacyjnych,
- wykonanie eksperymentów, ich ocena oraz sformułowanie wniosków końcowych.

1.3. Przegląd literatury

Część pracy dotycząca teoretycznych pojęć związanych z klasycznymi i głębokimi sieciami neuronowymi została oparta na kilku pozycjach obszernie opisujących tę tematykę: [3, 4, 5, 6].

Druga część opisująca praktyczne zastosowanie głębokich sieci neuronowych została oparta głównie na wiedzy zdobytej z książki [3] oraz wielu poradników i dokumentacji technicznej użytych narzędzi. Badania odnośnie doboru modelu najlepiej dopasowanego do rozwiązywania wybranego problemu zostały podparte wiedzą uzyskaną z licznych publikacji naukowych, które opisywały próby rozwiązania podobnych problemów. Są to m.in. pozycje [7, 8, 9, 10].

1.4. Struktura pracy

Tekst pracy podzielony został na dwie części. Pierwszą z nich stanowią rozdziały 2 oraz 3, które zawierają informacje teoretyczne. W rozdziale 2 opisane zostały zagadnienia głębokich oraz klasycznych sieci neuronowych wraz z ich porównaniem. Rozdział 3 przedstawia opis wybranego problemu z zakresu diagnostyki medycznej, dla którego zostanie podjęta próba rozwiązania przy pomocy głębokich sieci neuronowych.

Druga część składa się z rozdziałów 4 oraz 5. Rozdział 4 zawiera przegląd dostępnych bibliotek oraz narzędzi programistycznych. Natomiast w rozdziale 5 znalazł się opis implementacji procesu uczenia oraz przeprowadzonych badań w celu sprawdzenia możliwości rozwiązania wybranego problemu.

Rozdział 2

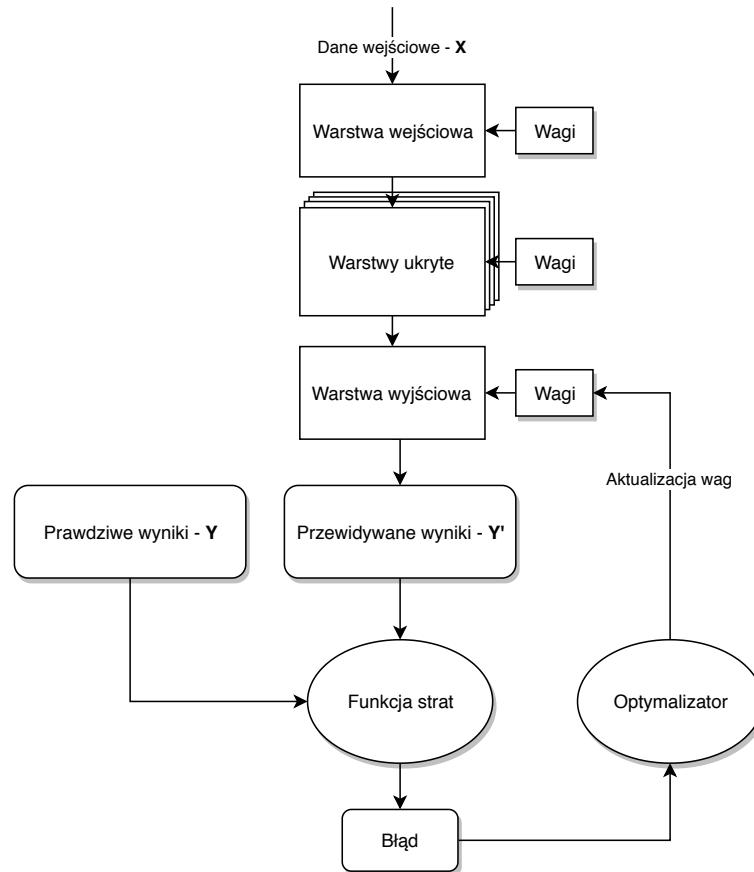
Wprowadzenie do głębokich sieci neuronowych

2.1. Deep learning, czyli głębokie sieci neuronowe

Deep learning to dziedzina uczenia maszynowego, która zakłada nowe podejście do nauki wzorców w danych. Kładzie ona nacisk przede wszystkim na wielowarstwową naukę reprezentacji danych, czyli tzw. map cech (ang. *feature maps*). Słowo *głębokie* (ang. *deep*) odnosi się głównie do ilości warstw, które składają się na model sieci. Ta cecha jest zwykle nazywana głębokością sieci (ang. *depth*). Nowoczesne modele głębokich sieci neuronowych często składają się z dziesiątek, a czasem nawet setek warstw. Każda z nich uczy się samodzielnie na podstawie przedstawionych im informacji.

Inne podejścia, takie jak np. uczenie maszynowe oraz klasyczne sieci neuronowe skupiają się na nauce znacznie mniejszej ilości reprezentacji danych, dlatego tego typu metody są często nazywane *płytkim uczeniem* (ang. *shallow learning*). W *deep learning*'u prawie zawsze używa się sieci neuronowych, które składają się z nałożonych na siebie wielu warstw neuronów. Głębokie sieci neuronowe działają więc pod wieloma względami identycznie jak sieci klasyczne (patrz rysunek 2.1). Różnicę natomiast stanowi ich wielkość (głębokość), używane metody umożliwiające efektywne uczenie dużych struktur danych oraz nowe architektury sieci.

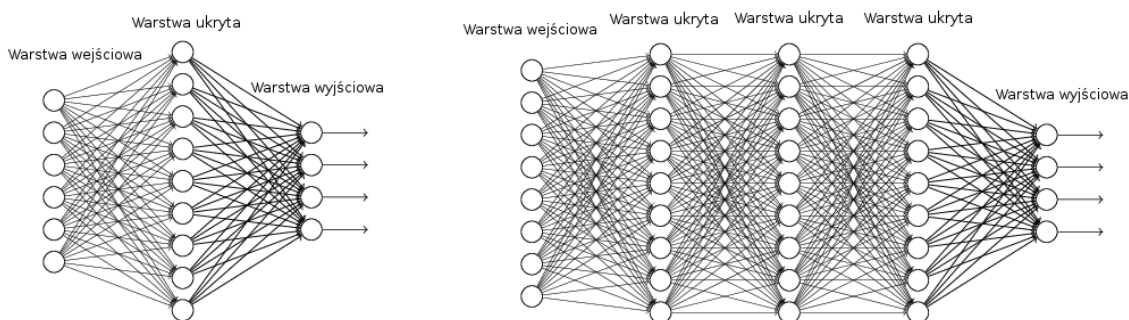
Klasyczne sieci neuronowe swego czasu straciły na popularności, gdyż ciężko było je wykorzystać do bardziej złożonych problemów z powodu ograniczeń ich wielkości. Sieci o większych rozmiarach były nieefektywne, gdyż czasy ich nauki były nieakceptowalne. Możliwość rozwoju głębokich sieci neuronowych była więc uwarunkowana powstaniem sprzętu o odpowiedniej mocy obliczeniowej, która umożliwiłaby wykonywanie tak dużej ilości obliczeń w skończonym czasie. Obecnie używane są do tego procesory kart graficznych (*GPU* - ang. *graphics processing unit*), które swoimi możliwościami znacznie przewyższają zwykłe procesory (*CPU* - *central processing unit*).



Rys. 2.1. Schemat procesu uczenia głębokiej sieci neuronowej

2.2. Porównanie klasycznych oraz głębokich sieci neuronowych

Głębokie sieci neuronowe w swoim działaniu bardzo przypominają sieci klasyczne. Wynika to z faktu, gdyż tak na prawdę niewiele się od nich różnią. Sieć można nazwać głęboką, gdy posiada ona wystarczająco dużo warstw ukrytych. Ilość warstw klasyfikująca sieć jako głęboką nie jest jednak ustalona. Zwykle przyjmuje się, że klasyczne, płytkie sieci neuronowe posiadają maksymalnie jedną warstwę ukrytą. Różnice w budowie klasycznych oraz głębokich sieci neuronowych ilustruje rysunek 2.2.



Rys. 2.2. Porównanie struktury sieci płytkiej oraz głębokiej

Głębokie sieci neuronowe nie ograniczają się jednak do stosowania tej jednej, podstawowej struktury, która składa się z warstw neuronów połączonych każdy z każdym (ang. *fully connected*). Istnieją również inne, bardziej rozbudowane architektury sieci neuronowych, które zostaną opisane w rozdziale 2.3.

Zaletą klasycznych sieci neuronowych jest niewątpliwie ich prostota oraz szybkość nauki. Często jednak ich zastosowanie w bardziej złożonych problemach bywa nieefektywne. Przewagą głębokich sieci neuronowych jest niewątpliwie ich zdolność do rozwiązywania o wiele bardziej skomplikowanych problemów. Przekłada się to oczywiście na dłuższe czasy nauki, jednak biorąc pod uwagę aktualnie dostępny sprzęt z dużą mocą obliczeniową, są one akceptowalne.

2.3. Architektury głębokich sieci neuronowych

Wyróżnia się 5 najczęściej stosowanych architektur głębokich sieci neuronowych:

- **Sieci splotowe** (*CNN* - ang. *Convolutional Neural Networks*),
- **Rekurencyjne sieci neuronowe** (ang. *Recurrent Neural Networks*),
- **ResNets** (*Residual Networks*),
- **Autoenkodery** (ang. *Autoencoders*),
- **GAN** (ang. *Generative Adversarial Networks*).

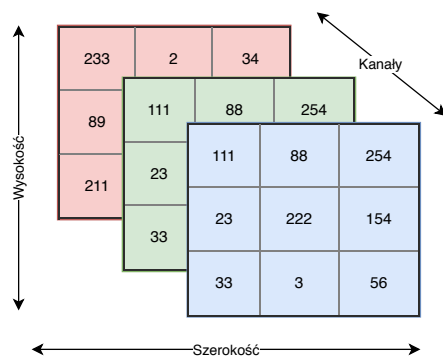
Zostaną omówione jedynie dwie pierwsze architektury, gdyż to one znalazły zastosowanie w niniejszej pracy.

2.3.1. Splotowe sieci neuronowe

Splotowe sieci neuronowe są niewątpliwie najpopularniejszą architekturą głębokich sieci neuronowych. Są one przeważnie używane do zadań związanych z rozpoznawaniem obrazów, jednak równie dobrze sprawdzają się w wielu innych dziedzinach.

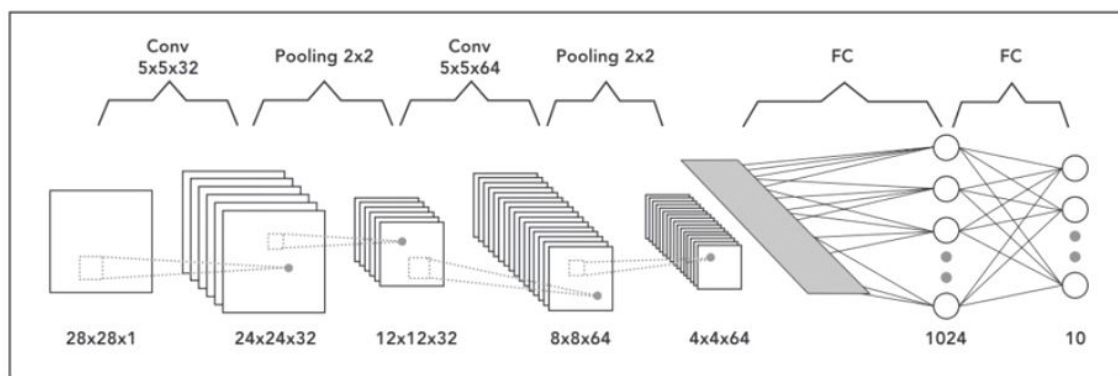
Przewagą tego typu sieci nad klasycznymi sieciami neuronowymi jest możliwość rozpoznawania tymczasowych oraz przestrzennych zależności pomiędzy danymi. Jest to możliwe, dzięki temu, że dane które trafiają do sieci mogą posiadać strukturę wielowymiarową. Przykładowo, obrazki zapisane w formacie RGB są 3-wymiarową macierzą przechowującą informację o wartościach koloru każdego z pikseli (rysunek 2.3).

Na danych przestrzennych wykonywane są operacje takie jak: konwolucja (ang. *convolution*), zmniejszanie rozmiaru danych (ang. *subsampling*) oraz aktywacja (ang. *activation*). Znalezione cechy są następnie podawane na wejście klasycznej sieci, gdzie neurony połączone są każdy z każdym (ang. *fully connected*), która ostatecznie przetwarza dane i zwraca wynik końcowy. Działanie sieci splotowej polega na tworzeniu kolejnych tzw. map cech (ang. *feature map*), które są abstrakcyjnymi reprezentacjami danych umożliwiającymi wychwycenie podobieństw i zależności pomiędzy danymi. Szczegółowy opis sposobu działania tych operacji został przedstawiony m.in.



Rys. 2.3. Dane liczbowe obrazka w formacie RGB

w pozycjach [3] oraz [6]. Przykładowy schemat splotowej sieci neuronowej składający się z kilku warstw konwolucyjnych połączonych z warstwami zmniejszającymi rozmiar danych (tzw. *pooling*'iem), która została zakończona dwoma warstwami sieci *fully connected* (FC) przedstawiony jest na rysunku 2.4.



Rys. 2.4. Schemat działania sieci splotowej. Źródło: [1]

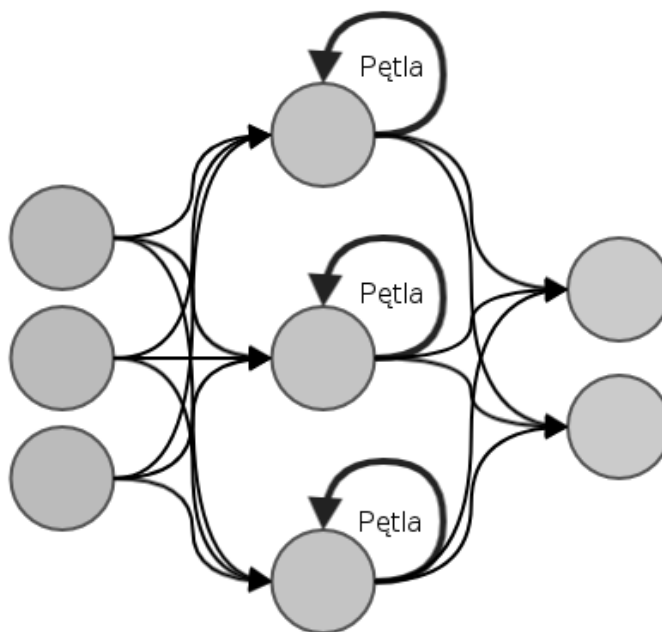
Na rysunku 2.4 można zauważyć, że wymiary danych po przejściu przez kolejne warstwy znacząco się zmieniają. Ich wielkość uzależniona jest od ilości oraz rozmiaru zastosowanych filtrów w warstwach konwolucyjnych, które wyszukują w danych zestawy cech. Przy zastosowaniu dużej ilości filtrów rozmiar danych ulega zwiększeniu. W celu ograniczenia ich wielkości stosowane są warstwy *pooling*'u, które zmniejszają ich rozmiar np. w *max pooling*'u wybierana jest największa wartość z danego obszaru. W ten sposób przygotowane cechy trafiają na warstwy *fully connected*, które zwrócą ostateczne wyniki.

Zaletą tego typu sieci jest bez wątpienia możliwość przetwarzania wielowymiarowych danych, dzięki czemu są one w stanie odnajdywać przestrzenne zależności pomiędzy nimi. Przekłada się to na ich wysoką skuteczność i wszechstronność.

2.3.2. Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe są często używane do rozwiązywania problemów, gdzie wykorzystywane są dane, w których znaczącą rolę pełni ich kolejność np. tekst lub szeregi czasowe. W praktyce znajdują one zastosowanie w takich problemach jak np. prognozowanie pogody czy analiza tekstu.

Idea działania rekurencyjnych sieci neuronowych polega na zapamiętywaniu pewnego ciągu sekwencji i dokonywanie wyborów nie tylko na podstawie aktualnych danych, ale również tych, które były przetwarzane poprzednio. Jest to realizowane za pomocą wewnętrznej pętli, która polega na przekazywaniu wyników z danej warstwy jako dodatkowych danych wejściowych podczas kolejnej iteracji. Schemat sieciadającej pętlę zaprezentowany został na rysunku 2.5.



Rys. 2.5. Schemat rekurencyjnej sieci neuronowej

Zaletą rekurencyjnych sieci neuronowych jest niewątpliwie to, że współdzielą one wagi, dzięki temu pozwala to zredukować ich ilość. Mogą być one również używane w połączeniu z sieciami splotowymi. Do wad należy m.in. to, że nie są one w stanie zapamiętywać zbyt długich sekwencji.

Rozdział 3

Omówienie wybranego problemu diagnostyki medycznej

3.1. Przedstawienie problemu

W celu sprawdzenia możliwości zastosowania głębokich sieci neuronowych do diagnostyki medycznej został wybrany problem **wykrywania napadów padaczki na podstawie odczytów z elektroencefalogramu (EEG)**.

Padaczka (zwana również epilepsją) to grupa przewlekłych zaburzeń neurologicznych, które charakteryzują się napadami padaczkowymi. Napady są powodowane przejściowymi zaburzeniami pracy mózgu polegającymi na samorzutnych wyładowaniach bioelektrycznych w komórkach nerwowych mózgu. Występują one ze zróżnicowaną siłą, od krótkich i ledwo zauważalnych, aż do długich, silnych wstrząsów. Może towarzyszyć im utrata świadomości oraz drgawki. Epilepsja jest nieuleczalna, jednak w wielu przypadkach może być kontrolowana lekami. Cierpi na nią ok. 1% ludzi na całym świecie, czyli ok. 65 milionów [11].

Podczas standardowego badania aktywność mózgu monitorowana jest za pomocą elektroencefalogramu. Odczyty fal mózgowych tworzone są na podstawie sygnałów odbieranych przez elektrody umieszczane na głowie pacjenta. Aktualnie odczyty muszą być analizowane przez wykwalifikowanego neurologa, który jest w stanie rozpoznać charakterystyczne wzorce wskazujące na występowanie choroby. Wizualna diagnostyka odczytów jest jednak niezwykle pracochłonna i wymaga specjalistycznej wiedzy. Skutkiem tego mogą być limity pacjentów, których badania mogą zostać przeanalizowane. Ograniczenia te mogą być rozwiązane poprzez próbę automatycznego wykrywania napadów padaczkowych przez maszynę.

Problem automatycznego wykrywania ataków był już wielokrotnie badany. Często do rozwiązania problemów używane były algorytmy, które korzystały ze specjalnie przygotowanych zestawów cech charakteryzujących ataki. Znanym faktem jest jednak to, że napady padaczki mogą być niezwykle zróżnicowane. Tyczy się to zarówno przypadków, w których porównywane są charakterystyki ataków różnych pacjentów, jak i tych, które występują podczas badania jednego chorego. Zostały podejmowane również próby wykorzystania głębokich sieci neuronowych, które byłyby w stanie wykryć w danych bardziej generyczne wzorce pozwalające na klasyfikację z większą dokładnością i odpornością na zróżnicowanie danych. Sposób, oraz wyniki przeprowadzonych w ten sposób badań można znaleźć m.in. w pracach [7, 8, 9].

3.2. Dostępne dane

Badania opisane w niniejszej pracy zostały przeprowadzone na danych udostępnionych przez Szpital Uniwersytecki w Zielonej Górze. Dostępne są odczyty 104 chorych pacjentów w różnym wieku, każde trwające około 15 minut. Zostały one przeanalizowane przez neurologa, który oznaczył punkty w czasie, w których doszło do ataków.

Odpowiednio przygotowane odczyty wraz z wynikami badań posłużą jako dane, na podstawie których zostaną przeprowadzone eksperymenty polegające na próbie nauki głębokiej sieci neuronowej do automatycznego rozpoznawania stanów padaczkowych.

3.3. Oczekiwane rezultaty

Wymienione publikacje naukowe przedstawiają obiecujące wyniki. Niektóre z nich prezentują rezultaty na poziomie nawet ok. 90% skuteczności. Badania te zostały jednak przeprowadzone na zupełnie innych danych. Często używana jest ogólnie dostępna baza CHB-MIT (Children's Hospital of Boston-Massachusetts Institute of Technology). Nie są znane również szczegóły implementacyjne, dlatego opracowane zostanie własne rozwiązanie analizowanego problemu.

Rezultatem oczekiwanym w tej pracy jest osiągnięcie wyników świadczących o tym, że technologię deep learning'u oraz zaproponowane rozwiązanie można z powodzeniem zastosować również do posiadanej bazy odczytów. Jako próg przyjęto **70%** skuteczności.

Rozdział 4

Przegląd dostępnych bibliotek oraz narzędzi programistycznych

4.1. Dostępne narzędzia

Istnieje bardzo dużo narzędzi umożliwiających tworzenie oraz naukę głębokich sieci neuronowych. Do tych najbardziej popularnych zalicza się m.in.:

- **TensorFlow** - najczęściej używana biblioteka do deep learning'u stworzona przez Google. Jest ona używana m.in. przez takie firmy jak Airbus, Twitter oraz IBM. Google wykorzystuje ją również w swoich produktach np. tłumacz Google zbudowany jest na bazie TensorFlow. Rekomendowanym językiem jest Python, jednak istnieje możliwość używania języków Java, C oraz Go.
- **Keras** - biblioteka dostarczająca wysokopoziomowe API umożliwiające tworzenie sieci neuronowych w prosty sposób. Keras sam w sobie nie posiada silnika umożliwiającego tworzenia sieci neuronowych, jednak używa jednej z wybranych bibliotek (TensorFlow, CNTK lub Theano). Biblioteka jest napisana w Pythonie i ten język jest zalecany do użytku, jednak istnieje możliwość użycia również języka R.
- **Theano** - biblioteka dedykowana dla Pythona umożliwiająca tworzenie sieci neuronowych może być używana w połączeniu z biblioteką Keras.
- **Caffe** - biblioteka stworzona przez zespół Berkeley AI Research (BAIR). Została napisana w C++, jednak umożliwia również korzystanie z Matlab'a oraz Python'a.
- **Deeplearning4j** - biblioteka napisana w Javie, kompatybilna ze wszystkimi językami uruchamianymi na wirtualnej maszynie Javy (JVM) m.in. Java, Groovy, Scala czy Kotlin. Jest bardziej ukierunkowana na użycie w aplikacjach biznesowych, niż naukowo-badawczych.

Wszystkie wymienione biblioteki są dostępne za darmo.

4.2. Wybrany stos technologiczny

Stos technologiczny wybrany do wykonania badań w tej pracy to:

- **Python** - najbardziej popularny język programowania do deep learning'u, bardzo dobrze przystosowany do zastosowań naukowych.
- **Keras/TensorFlow** - najczęściej wykorzystywany zestaw bibliotek posiadający bardzo dobrą dokumentację, opisywany w wielu poradnikach. Wspiera również możliwość wykonywania obliczeń na karcie graficznej. Dostarcza dodatkowy zestaw narzędzi umożliwiających monitorowania sieci m.in. TensorBoard.
- **Nvidia cuDNN** - biblioteka umożliwiająca wykonywanie obliczeń na karcie graficznej podczas nauki sieci neuronowych. Jej używanie jest bardzo zalecane, gdyż umożliwia o wiele szybsze przetwarzanie danych. Wymagane jest posiadanie odpowiedniej karty graficznej wspierającej tę technologię.
- **Jupyter Notebook** - środowisko programistyczne umożliwiające pracę w językach Julia, Python oraz R.
- **Docker** - narzędzie służące do konteneryzacji. Umożliwia uruchamianie całego środowiska w odizolowanych kontenerach za pomocą dostępnych publicznie obrazów bez konieczności instalacji w systemie żadnych dodatkowych narzędzi.

Rozdział 5

Rozwiązanie problemu

Proces uczenia sieci neuronowej stworzony został w środowisku Jupyter Notebook. Umożliwia ono tworzenie dokumentów zawierających opisy, wykresy oraz wykonywalne kody źródłowe. Nazwa Jupyter wzięła się z tego, że środowisko umożliwia pracę w trzech językach: Julia, Python oraz R.

Dokument składa się z wielu komórek, które mogą być wywoływane niezależnie. Przewagą kodu pisanego w tym środowisku nad tradycyjnymi skryptami jest oferowana przez nie możliwość zapamiętywania stanu zmiennych. Dzięki temu nie ma potrzeby każdorazowego wywoływania całego kodu, a jedynie fragmentów, które w danej chwili są potrzebne np. jednorazowe przygotowanie danych, a następnie wykonywanie jedynie procesu uczenia.

Proces uczenia składa się z kilku kroków:

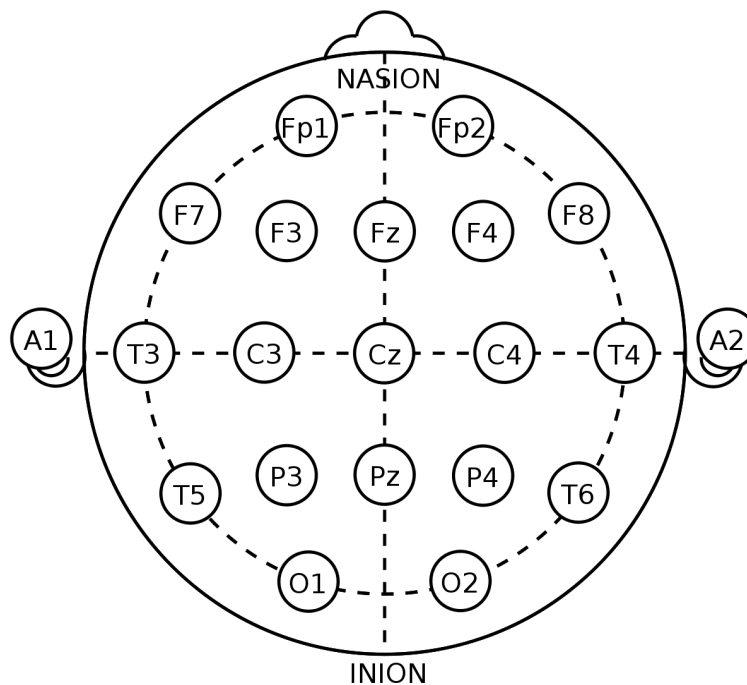
- pobranie i przygotowanie danych za pomocą przygotowanych skryptów 5.1,
- podział danych na zbiory służące do nauki oraz testowania parzy pomocy k-krotnej walidacji krzyżowej (*ang. k-fold cross-validation*) 5.2,
- utworzenie modelu sieci neuronowej 5.3.1,
- rozpoczęcie uczenia modelu 5.3,
- sprawdzenie skuteczności i przedstawienie wyników 5.3.4.

Wszystkie wymienione kroki przedstawiają pełen proces uczenia, który pokazuje z jaką dokładnością model jest w stanie wykrywać stany padaczkowe.

5.1. Przygotowanie danych

W celu wykorzystania dostępnych danych w procesie uczenia sieci neuronowej należy je najpierw odpowiednio przygotować. Dostarczone zostały dane w postaci plików tekstowych z liczbami reprezentującymi wartości zmierzone przy pomocy elektroencefalogramu. W poszczególnych kolumnach znajdują się wartości różnic potencjałów pomiędzy dwoma elektrodami dla 16 kanałów. Jest to bardzo typowy układ wynikający z rozmieszczenia elektrod na głowie pacjenta - tzw. system 10-20 (patrz rysunek 5.1).

Nagłówki kolumn poszczególnych kanałów posiadają etykiety *EEG_FP1_F3*, *EEG_FP2_F4* itd. Przykładowo, *EEG_FP1_F3* oznacza sygnał różnicowy pomiędzy elektrodami *FP1* oraz *F3*. Ostatnia kolumna (o nagłówku *t*) przedstawia czas w sekundach, w którym miał miejsce pomiar.



Rys. 5.1. Rozmieszczenie elektrod w systemie 10-20. Źródło: [2]

Dostępne są dane 104 pacjentów, u których wystąpiły ataki. Każdy z nich posiada odczyty wykonane z częstotliwością 500Hz przez blisko 15 minut. Jest to około 450 000 linii w każdym pliku. Dane zajmują 6,5 GB. Fragment jednego z plików przedstawiony został na listingu 5.1:

Listing 5.1. Dane liczbowe z odczytu EEG

```

1 "EEG_FP1_F3" "EEG_FP2_F4" "EEG_F3_C3" "EEG_F4_C4" "EEG_C3_P3" "EEG_C4_P4" "EEG_P3_01" "EEG_P4_02" "EEG_FP1_F7" "EEG_FP2_F8" "EEG_F7_T3"
  "EEG_F8_T4" "EEG_T3_T5" "EEG_T4_T6" "EEG_T5_01" "EEG_T6_02" "t"
2 -1.7032 -3.3727 5.9014 2.512 2.6404 4.0193 16.9162 35.4401 5.3506 1.1355 -1.0843 -3.9932 9.4594 14.8409 10.0322 26.6143 0
3 -3.1702 -4.5883 5.8762 2.9638 3.5952 3.9019 17.1137 36.3634 4.3875 0.543 -1.356 -4.5697 10.309 14.1082 10.0724 28.5593 0.002
4 -4.2669 -5.2157 5.6852 3.1698 4.0053 3.6091 16.4135 36.7024 2.9881 0.4522 -1.7584 -5.4206 11.1134 13.3778 9.4926 29.8509 0.004
5 -4.8106 -5.2287 5.2889 2.6745 3.714 3.0257 14.9097 36.4086 1.6189 0.2802 -2.1206 -6.4522 11.2876 12.7727 8.3198 30.2831 0.006
6 -5.0684 -5.1176 4.7665 1.7354 2.9116 2.354 12.7327 35.6144 0.5718 -0.618 -2.57 -7.3378 10.6144 12.2487 6.7256 30.2934 0.008
7 -5.3637 -5.1568 4.399 1.03 1.764 1.9509 10.2011 34.6652 -0.2974 -2.1804 -3.1603 -7.7025 9.3368 11.727 5.1179 30.6433 0.01
8 -5.6214 -5.0654 4.3918 0.9508 0.24 1.8783 7.7188 33.8484 -1.2102 -3.7284 -3.7707 -7.4767 7.7603 11.2122 3.9518 31.6004 0.012
9 -5.3918 -4.3726 4.6296 1.2678 -1.7391 1.9194 5.4296 32.8928 -2.0525 -4.7365 -4.371 -6.9662 6.0031 10.7833 3.3565 32.6295 0.014
10 -4.4216 -2.9806 4.8926 1.4897 -4.0924 1.8759 3.1044 31.2302 -2.5055 -5.2621 -5.1323 -6.5737 4.1319 10.4842 2.993 32.964 0.016
11 -3.0671 -1.3206 5.1772 1.4303 -6.5959 1.8178 0.4606 28.7023 -2.4988 -5.6443 -6.219 -6.4834 2.3209 10.2523 2.3732 32.5009 0.018
12 -2.0125 0.0126 5.624 1.3708 -9.0008 1.8601 -2.4571 25.9613 -2.4216 -6.0744 -7.4599 -6.5563 0.7207 10.0158 1.3118 31.8217 0.02
13 -1.6282 0.6792 6.1788 1.668 -11.1278 1.9872 -5.285 23.879 -2.7505 -6.5044 -8.4291 -6.466 -0.7268 9.7492 0.0498 31.4306 0.022
14 -1.6657 0.8034 6.6003 2.2822 -12.8962 2.0405 -7.673 22.7264 -3.4955 -6.877 -8.9121 -5.952 -2.1528 9.4663 -1.074 31.2145 0.024
15 -1.5719 0.7576 6.7409 2.7736 -14.344 1.9775 -9.5717 21.9936 -4.0627 -7.2927 -9.1401 -5.0143 -3.5228 9.1023 -2.0194 30.7051 0.026
16 -1.0657 0.7511 6.7409 2.833 -15.5319 1.8759 -11.2415 20.9346 -3.915 -7.9234 -9.4655 -3.9029 -4.6154 8.5574 -3.1008 29.6605 0.028
17 -0.4658 0.64 6.7481 2.6388 -16.4105 1.8674 -12.9338 19.1106 -3.3982 -8.8312 -9.7774 -2.8922 -5.2413 7.7923 -4.6393 28.1837 0.03
18 -0.4283 0.1433 6.6724 2.5675 -16.8789 1.9739 -14.6844 16.5569 -3.66 -9.9301 -9.5996 -2.0969 -5.4112 6.8556 -6.6483 26.4136 0.032
19 -1.2579 -0.837 6.2977 2.6904 -17.0425 2.1591 -16.2868 13.5512 -5.3513 -11.1628 -8.8048 -1.3918 -5.2607 5.7612 -8.8668 24.3656 0.034
20 -2.5281 -2.1571 5.5879 2.726 -17.2241 2.3092 -17.4808 10.3421 -7.6669 -12.6201 -7.9998 -0.5653 -4.9359 4.4372 -11.0386 21.9677 0.036
21 -3.4233 -3.5492 4.6692 2.5041 -17.6387 2.3055 -18.1227 6.9393 -9.0697 -14.3688 -7.8724 0.5427 -4.5573 2.8026 -13.0119 19.2251 0.038
22 -3.5686 -4.719 3.6748 2.1871 -18.0668 2.0102 -18.226 3.2427 -9.0093 -16.2083 -8.3252 1.8902 -4.2347 0.9176 -14.6128 16.1274 0.04
23 -3.292 -5.4248 2.7057 1.9731 -18.0892 1.3869 -17.9791 -0.8025 -8.4321 -17.699 -8.5968 3.1857 -3.9831 -0.9698 -15.6362 12.613 0.042
24 -3.1795 -5.5751 2.0283 1.7869 -17.5715 0.4949 -17.5975 -5.0931 -8.516 -18.5016 -8.1441 4.0158 -3.6433 -2.5673 -16.0108 8.6663 0.044

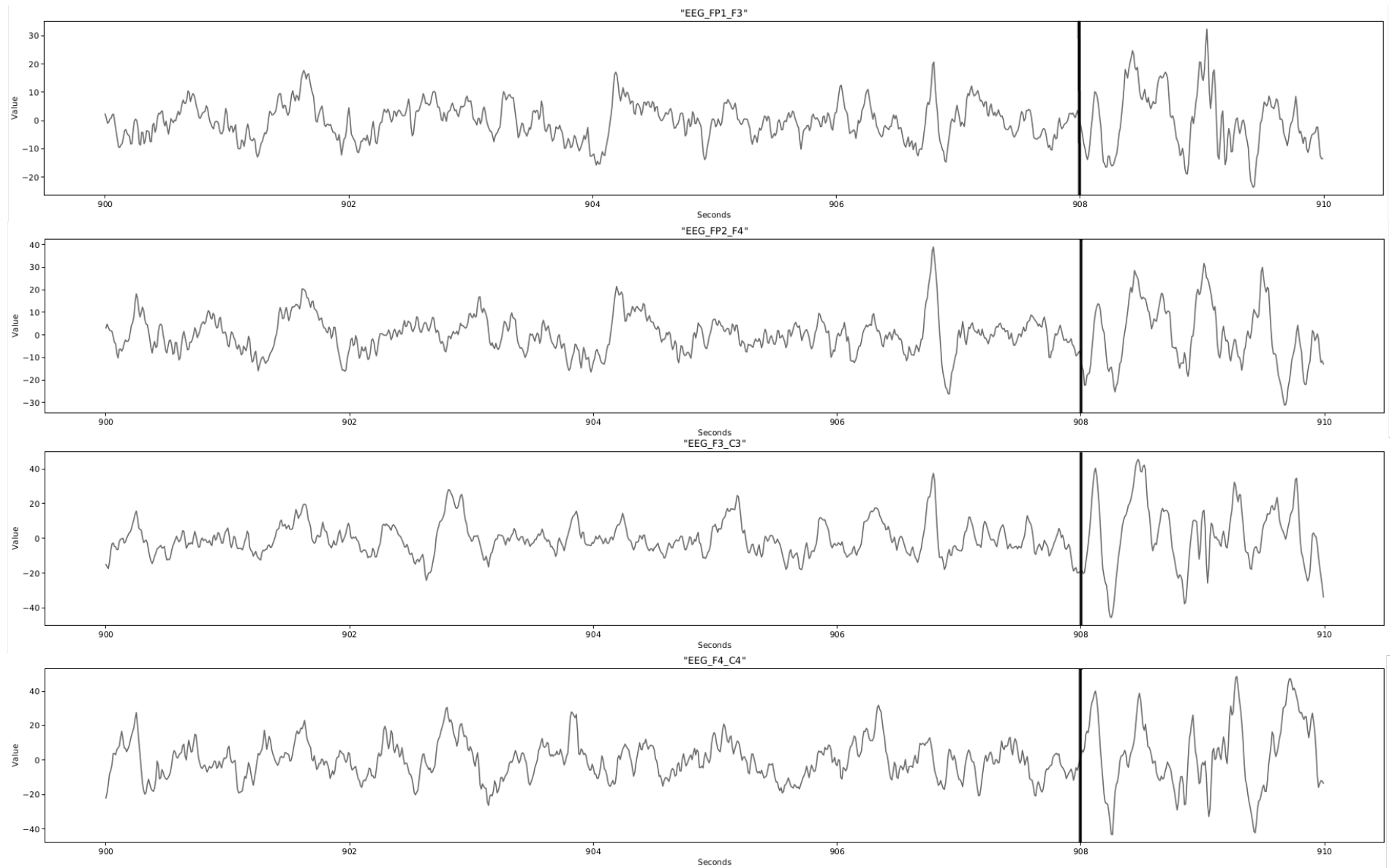
```

Dodatkowo dostarczony został plik przechowujący wyznaczone przez lekarza momenty wystąpienia ataków dla każdego z pacjentów. Plik w kolejnych wierszach zawiera punkty w czasie rozdzielone przecinkami (w sekundach). Każdy wiersz odpowiada jednemu pacjentowi, dlatego plik zawiera 104 wiersze. Fragment pliku z czasami wystąpienia ataków:

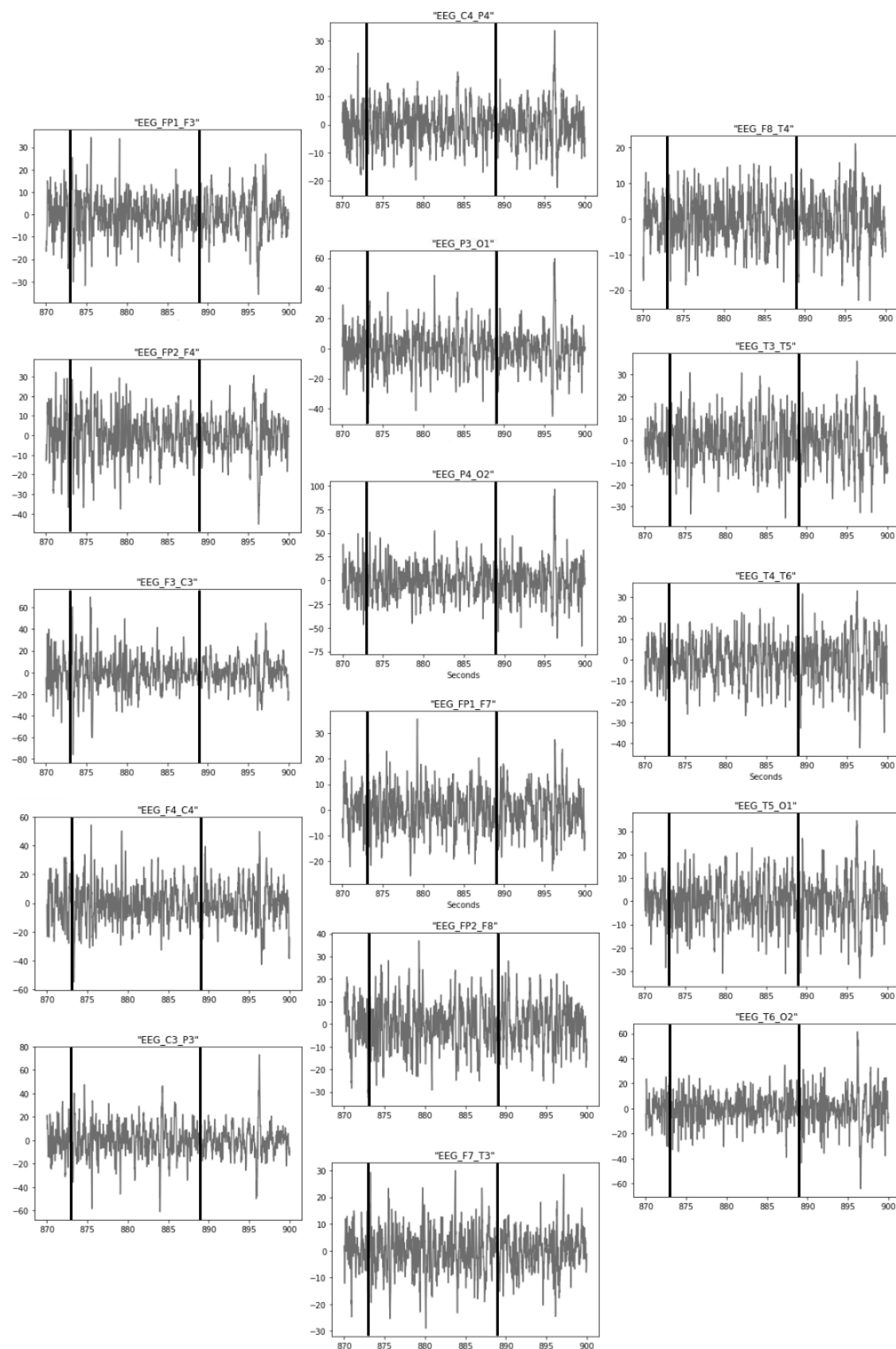
Listing 5.2. Punkty w czasie wystąpienia ataków (w sekundach)

```
1 835,853,865,873,889,908
2 18,48,110,309,466,618,757
3 216,239
4 44,329,501,559,622
5 36,190,406,576,714,754
6 158,510,917
7 622,653,676,737
```

Dane z kilku kanałów z naniesionym momentem wystąpienia ataku przedstawione zostały na rysunku 5.2. Można zaobserwować, że bez posiadania specjalistycznej wiedzy ciężko byłoby określić moment wystąpienia ataku. Fragmenty odczytu w momencie ataku na pierwszy rzut oka nieznacznie różnią się od pozostałych. Świadczy o wiele większej złożoności problemu w porównaniu do zadań, z którymi mózg praktycznie każdego człowieka radzi sobie doskonale np. rozpoznawanie liczb, odróżnianie psów od kotów itp. Wykresy przedstawiające odczyty ze wszystkich 16 kanałów zaprezentowano na rysunku 5.3.



Rys. 5.2. Wykres przedstawiający odczyt EEG z zaznaczonym momentem wystąpienia ataku



Rys. 5.3. Odczyty ze wszystkich 16 kanałów z zaznaczonymi miejscami wystąpienia ataków

W celu wykorzystania przedstawionych danych w procesie uczenia sieci neu-

ronowej muszą zostać one odpowiednio przygotowane. Wybrane podejście zakłada podzielenie danych na okna czasowe o określonej długości. Znane są jedynie początki ataków, jednak nie wiadomo jak długo trwały. Długość okna czasowego, według którego podzielone zostaną dane będzie więc jednym z parametrów, który należy dobrać w celu uzyskania najlepszych rezultatów.

Do przygotowania danych stworzony został skrypt, który wykonuje następujące czynności:

- wczytanie danych pacjentów oraz informacji o atakach,
- przetworzenie danych do częstotliwości 100Hz,
- podział danych na okna czasowe o długości przekazanej jako parametr (w sekundach),
- normalizacja danych - średnia 0, odchylenie standardowe 1,
- konwersja danych dotyczących czasu wystąpienia ataków na postać tzw. "jeden z n" (*ang. one-hot encoding*),
- wyświetlanie informacji o postępie przetwarzanych plików,
- zapis pobranych danych do plików tymczasowych umożliwiających szybszy odczyt.

Technicznie skrypty zostały podzielone na 2 pliki:

- `data_reader.py`,
- `chunks_creator.py`.

Pierwszy z nich zajmuje się wczytywaniem i korzysta z drugiego w celu stworzenia okien czasowych. Główną funkcją dostarczaną przez skrypt, która umożliwia wykonanie wszystkich wymienionych wyżej czynności jest `get_data()`. Jako parametr przyjmuje ona czas w sekundach, który oznacza długość okna czasowego. Znajduje się ona na końcu pliku, gdyż z uwagi na specyfikę języka wszystkie wykorzystywane przez nią funkcje muszą być wcześniej zadeklarowane. Kod skryptu pobierającego dane rozszerzony o komentarze opisujące poszczególne kroki zaprezentowany został na listingu 5.3.

Listing 5.3. `data_reader.py`

```
1 import os
2 import numpy as np
3 import pickle
4
5 from chunks_creator import prepare_chunks
6 from chunks_creator import flatten_chunks
7
8 from sklearn.preprocessing import StandardScaler
9
10 INPUT_DATA_FILE_PATH='tmp/input-{}sec.pkl'
11
12 DATA_FREQUENCY = 500
13 SAMPLING_RATE = 5
14 FREQUENCY_TO_SAMPLING_RATIO = DATA_FREQUENCY // SAMPLING_RATE
15
16
```

```
17 # Konwertuje plik z danymi pacjenta z postaci tekstowej do tablicowej oraz zmienia
    częstotliwość próbkowania do 100Hz
18 def parse_file(file, sampling_rate):
19     lines = file.split('\n')
20     headers = lines[0].split('\t')
21     # to one before last because the last one is empty
22     data = lines[1:-1]
23
24     number_of_lines = len(data)
25
26     float_data = np.zeros((number_of_lines, len(headers)))
27     for line_number, line in enumerate(data):
28         values = [float(value) for value in line.split('\t')]
29         float_data[line_number, :] = values
30
31     return float_data[::sampling_rate], headers
32
33
34 # Wczytuje dane pacjentów z dysku
35 def read_input_files(end, data_path, sampling_rate):
36     input_path = os.path.join(data_path, 'input_500Hz/sick')
37     input_file_names = os.listdir(input_path)
38     input_file_names.sort(key=int)
39
40     start = None
41
42     files_content = []
43     for file_name in input_file_names[start:end]:
44         file_path = os.path.join(input_path, file_name)
45         file = open(file_path, 'r')
46         (columns, headers) = parse_file(file.read(), sampling_rate)
47         print('Loaded input file:', file_name)
48         file.close()
49         files_content.append(columns)
50     print('--Input files loaded--')
51     return files_content, headers
52
53
54 def create_target_index(value, frequency_to_sampling_ratio):
55     value = int(value)
56     return int(value * frequency_to_sampling_ratio)
57
58
59 # Odczytuje informacje dotyczące czasów ataków i konwertuje do postaci one-hot
60 def read_target_files(end, data_path, sampling_rate, data_frequency):
61     frequency_to_sampling_ratio = data_frequency // sampling_rate
62     targets_path = os.path.join(data_path, 'targets')
63     targets_file_name = os.listdir(targets_path)[0]
64     targets_file_path = os.path.join(targets_path, targets_file_name)
65
66     file = open(targets_file_path, 'r')
67     targets_content = file.read()
68     file.close()
69
70     lines = targets_content.split('\n')[:-1]
71     targets = []
72     for number, line in enumerate(lines, 1):
73         targets.append([(int(value), create_target_index(value,
74             frequency_to_sampling_ratio)) for value in line.split(',')])
74     print('--Target files loaded--')
75     return targets[:end]
76
77
78 # Pobiera dane pacjentów oraz czasy ataków
79 def read_data(data_path, sampling_rate, data_frequency, end=104):
80     (input_data, headers) = read_input_files(end, data_path, sampling_rate)
81     targets_data = read_target_files(end, data_path, sampling_rate, data_frequency)
82
83     return input_data, targets_data, headers
84
85
86 # Odczytuje dane i zapisuje zmienne do pliku tymczasowego
87 def load_data_to_file(chunk_size_in_seconds):
```

```

88     (input_data, target, headers) = read_data(data_path='data',
89                                               sampling_rate=SAMPLING_RATE,
90                                               data_frequency=DATA_FREQUENCY)
91
92     with open(INPUT_DATA_FILE_PATH.format(chunk_size_in_seconds), 'wb') as
93         input_variable_file:
94         pickle.dump([input_data, target, headers], input_variable_file)
95
96     del input_data, target, headers
97
98 # Normalizuje dane używając obiektu StandardScaler(). Zwrócone dane posiadają średnią 0
99   oraz odchylenie standardowe 1.
100 def normalize(x, y):
101     scalers = {}
102     for channel_number in range(x.shape[1]):
103         scalers[channel_number] = StandardScaler()
104         x[:, channel_number, :] = scalers[channel_number].fit_transform(x[:,
105                                     channel_number, :])
106     return x, y.astype(int)
107
108 # Pobiera zmienne z danymi z utworzonego pliku tymczasowego
109 def load_input_data(chunk_size_in_seconds):
110     with open(INPUT_DATA_FILE_PATH.format(chunk_size_in_seconds), 'rb') as
111         input_data_file:
112         input_data, target, headers = pickle.load(input_data_file)
113
114     return input_data, target, headers
115
116 # Pobiera dane wykorzystując funkcję load_input_data() i wywołuje funkcję
117   prepare_chunks() z pliku chunks_creator.py w celu utworzenia okien czasowych z
118   danymi. Utworzone porcje danych są następnie poddawane normalizacji.
119 def prepare_data(chunk_size_in_seconds):
120     input_data, target, headers = load_input_data(chunk_size_in_seconds)
121
122     chunks_input, chunks_target = prepare_chunks(input_data,
123                                                  target,
124                                                  chunk_size_in_seconds=chunk_size_in_seconds,
125                                                  ratio=FREQUENCY_TO_SAMPLING_RATIO)
126
127     x, y = flatten_chunks(chunks_input, chunks_target)
128     x, y = normalize(x, y)
129
130     return x, y
131
132 # Główna funkcja skryptu, która zwraca odpowiednio przygotowane dane. Przy pierwszym
133   uruchomieniu wczytuje dane z dysku do zmiennych i zapisuje je do pliku
134   tymczasowego. Odczyt zmiennych z danymi z pliku binarnego umożliwia szybszy dostęp
135   przy kolejnych uruchomieniach, gdyż otwieranie dużych plików tekstowych jest
136   czasochłonne.
137 def get_data(chunk_size_in_seconds):
138     file_exists = os.path.isfile(INPUT_DATA_FILE_PATH.format(chunk_size_in_seconds))
139
140     if not file_exists:
141         load_data_to_file(chunk_size_in_seconds)
142
143     return prepare_data(chunk_size_in_seconds)

```

Funkcja *prepare_data()* w powyższym skrypcie korzysta z osobnego pliku o nazwie *chunks_creator.py*, w którym zostały zdefiniowane funkcje tworzące okna czasowe z danymi. Dla danych każdego z pacjentów tworzonych jest $2 * n$ okien czasowych, gdzie n oznacza ilość zdiagnozowanych ataków.

Porcje danych zawierające ataki tworzone są na początku każdego z nich i trwają przez określony czas podany w parametrze. Następnie tworzone jest n kolejnych porcji danych zawierających dane liczbowe z okresu, w którym nie wystąpił atak. Utworzone okna czasowe zawierają więc taką samą ilość danych z atakami oraz bez.

Dodatkowo tworzona jest tablica zawierająca informacje o tym czy dla danego okna czasowego wystąpił (*wartość 1*) lub nie wystąpił (*wartość 0*) atak.

Implementacja skryptu odpowiadającego za przetwarzanie danych do postaci okien czasowych przedstawiona została na listingu 5.4.

Listing 5.4. chunks_creator.py

```

1  import random
2  import numpy as np
3
4
5  # Tworzy porcje danych, w których wystąpiły ataki.
6  def create_chunks_with_seizures(patient_data, seizure_seconds, chunk_size):
7      number_of_chunks = len(seizure_seconds)
8
9      chunks_input = np.zeros((number_of_chunks, chunk_size, 17))
10     chunks_target = np.zeros(number_of_chunks)
11
12     for seizure_number in range(0, number_of_chunks):
13         (seizure_time, seizure_index) = seizure_seconds[seizure_number]
14         chunk_start_index = seizure_index
15         chunk_end_index = chunk_start_index + chunk_size
16         chunks_input[seizure_number] = patient_data[chunk_start_index:chunk_end_index,
17             :]
18         # atak oznaczony wartością '1'
19         chunks_target[seizure_number] = 1
20
21     return (chunks_input, chunks_target)
22
23 # Sprawdza czy podany fragment znajduje się w zasięgu ataku.
24 def is_in_seizure_range(index, seizure_seconds, chunk_size):
25     for (seizure_time, seizure_index) in seizure_seconds:
26         seizure_start_index = seizure_index
27         seizure_end_index = seizure_start_index + chunk_size
28         if index in range(seizure_start_index, seizure_end_index):
29             return True
30
31     return False
32
33
34 # Tworzy początek pojedynczej porcji danych, który wybierany jest losowo, jednak
35     sprawdzane jest, aby nie zawierała ona momentów, w których wystąpił atak.
36 def create_non_seizure_data_start_index(data_size, chunk_size, seizure_seconds):
37     start_index = random.randint(0, data_size - chunk_size)
38
39     while (is_in_seizure_range(start_index, seizure_seconds, chunk_size)):
40         start_index = random.randint(0, data_size - chunk_size)
41
42     return start_index
43
44 # Tworzy porcje danych, w których nie wystąpiły ataki.
45 def create_chunks_without_seizures(patient_data, seizure_seconds, chunk_size):
46     number_of_chunks = len(seizure_seconds)
47
48     chunks_input = np.zeros((number_of_chunks, chunk_size, 17))
49     chunks_target = np.zeros(number_of_chunks)
50     (data_size, channels) = patient_data.shape
51
52     for chunk_number in range(0, number_of_chunks):
53         chunk_start_index = create_non_seizure_data_start_index(data_size, chunk_size,
54             seizure_seconds)
55
56         chunk_end_index = chunk_start_index + chunk_size
57         chunks_input[chunk_number] = patient_data[chunk_start_index:chunk_end_index, :]
58         # brak ataku oznaczone wartością '0'
59         chunks_target[chunk_number] = 0
60
61     return (chunks_input, chunks_target)
62

```

```

63 # Przystosowuje ilość wymiarów danych do procesu uczenia sieci neuronowej.
64 def flatten_chunks(chunks_input, chunks_target):
65     train_input = []
66     train_target = []
67
68     for patient_number in range(0, len(chunks_input)):
69         patient_data = chunks_input[patient_number]
70         patient_targets = chunks_target[patient_number]
71         for chunk_number in range(0, len(patient_data)):
72             train_input.append(patient_data[chunk_number])
73             train_target.append(patient_targets[chunk_number])
74
75     train_input = np.array(train_input)
76     train_target = np.array(train_target)
77
78     train_input = train_input[:, :, :-1]
79
80     return train_input, train_target
81
82
83 # Główna funkcja skryptu zwracająca okna czasowe stworzone z danych pacjentów w postaci
84   wielowymiarowej tablicy oraz tablicę zawierającą informacje o wystąpieniu ataków.
85 def prepare_chunks(input, target, chunk_size_in_seconds, ratio):
86     chunk_size = chunk_size_in_seconds * ratio
87     chunks_input = []
88     chunks_target = []
89
90     for patient_number in range(0, len(input)):
91         patient_chunks_input = []
92         patient_chunks_target = []
93         seizure_seconds = target[patient_number]
94         patient_data = input[patient_number]
95         (seizure_chunks_input, seizure_chunks_target) =
96             create_chunks_with_seizures(patient_data, seizure_seconds, chunk_size)
97         patient_chunks_input.extend(seizure_chunks_input)
98         patient_chunks_target.extend(seizure_chunks_target)
99
100         (non_seizure_chunks_input, non_seizure_chunks_target) =
101             create_chunks_without_seizures(patient_data, seizure_seconds, chunk_size)
102         patient_chunks_input.extend(non_seizure_chunks_input)
103         patient_chunks_target.extend(non_seizure_chunks_target)
104
105         chunks_input.append(np.array(patient_chunks_input))
106         chunks_target.append(np.array(patient_chunks_target))
107
108     return np.array(chunks_input), np.array(chunks_target)

```

Przedstawione skrypty umożliwiają pobranie odpowiednio przygotowanych danych, które następnie mogą być użyte w procesie uczenia sieci neuronowej.

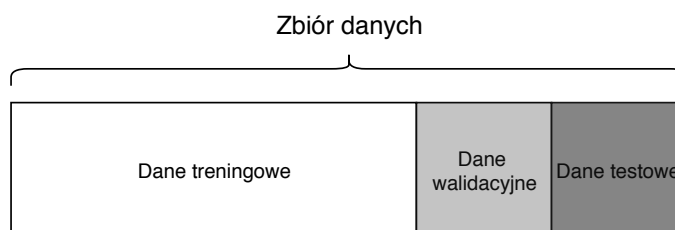
5.2. Metoda oceny wyników

Do zweryfikowania skuteczności modelu wymagane jest podzielenie danych na odpowiednie zbiory. Korzystanie z jednego, identycznego zbioru danych do nauki oraz testowania jest błędem, gdyż może wprowadzić złudne wrażenie, że model posiada bardzo dobrą skuteczność. Sieć neuronowa, której działanie weryfikowane jest na danych, które posłużyły do nauki będzie posiadać skuteczność bliską 100%. Dzieje się tak dlatego, że umie ona rozpoznawać te konkretne dane, gdyż zna dla nich wartości wyjściowe, które podane zostały podczas nauki. W ten sposób nauczony model będzie natomiast posiadał bardzo niską skuteczność, gdy na jego wejście podane zostaną dane, które nigdy wcześniej nie zostały mu przedstawione. W każdym rodzaju uczenia maszynowego dąży się do uzyskania jak największej zdolności generalizacji, czyli skuteczności klasyfikacji danych, które nie zostały nigdy wcześniej przekazane

do modelu. Przeciwnieństwem tego jest tzw. *nadmierne dopasowanie* (ang. *overfitting*), czyli sytuacja, w której model zbyt mocno dopasowuje się do danych uczących, co powoduje bardzo słabą zdolność generalizacji. Zjawisko to jest głównym problemem podczas uczenia modelu.

5.2.1. Hold-out

Podstawowym sposobem oceny wyników jest podzielenie danych na dwa zbiory: treningowy oraz testowy (ang. *hold-out*). Można dzielić je w różnych proporcjach np. 80/20, 90/10, 95/5 itp. w zależności od ilości posiadanych danych. Przy niewielkich zbiorach wydzielenie zbyt dużej liczby danych do zbioru testowego może powodować niedobory w procesie nauki. Wydzielenie jedynie dwóch zbiorów danych nie rozwiązuje jednak do końca problemu *overfitting*'u, gdyż zbyt częsta zmiana modelu i sprawdzanie jego skuteczności na danych testowych może spowodować zbyt duże dopasowanie do zbioru testowego [3]. Z tego względu często stosowany jest dodatkowy zbiór walidacyjny (ang. *validation set*). W tym podejściu model uczony jest na danych treningowych, sprawdzany na danych walidacyjnych i na podstawie tych wyników wprowadza się modyfikacje w modelu. Zbiór testowy służy jedynie do ostatecznego sprawdzenia skuteczności modelu, gdyż są to dane, które nigdy wcześniej nie zostały użyte w procesie nauki. Dane dzielone są w proporcjach np. 80/10/10, 90/5/5. Schemat tego podejścia przedstawiony jest na rysunku 5.4



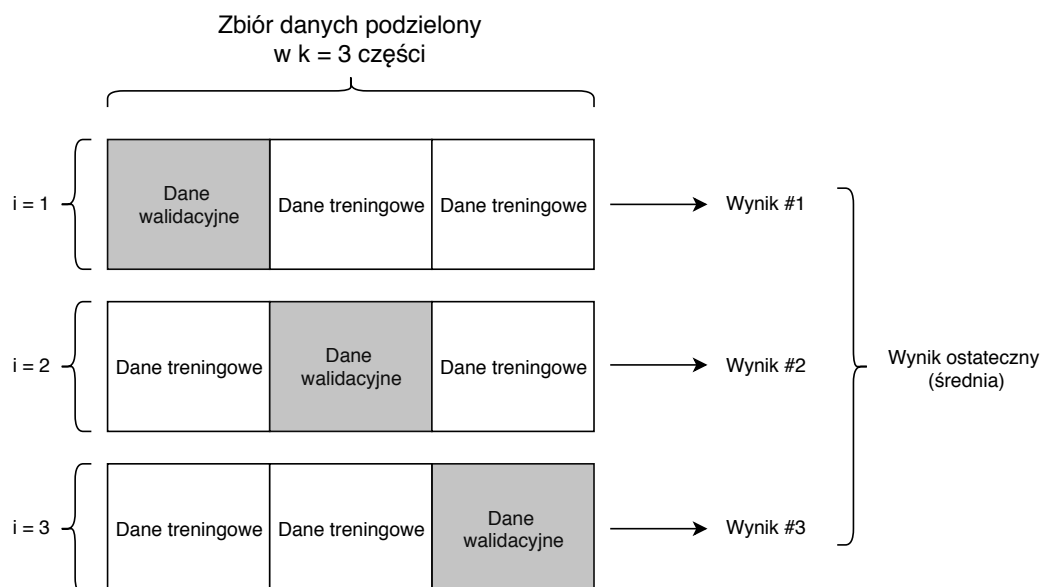
Rys. 5.4. Podział danych na zbiory: treningowy, walidacyjny i testowy

Metoda ta rozwiązuje problem zbyt dużego dopasowywania się do zbioru testowego, jednak również posiada wadę poprzedniego rozwiązania. Przy niewielkiej ilości danych wydzielanie osobnych zbiorów może spowodować, że zbiór uczący będzie zbyt mały. W celu uniknięcia tego problemu można zastosować tzw. walidację krzyżową i jej iteracyjne rozszerzenie.

5.2.2. Walidacja krzyżowa

Kolejną metodą walidacji jest tzw. *k-krotna walidacja krzyżowa* (ang. *k-fold cross-validation*). Tak samo jak w przypadku metody *holdout* należy najpierw wydzielić zbiór testowy, który nie będzie brał udziału w procesie uczenia i walidacji. Pozostałe dane dzielone są na k zbiorów, przeważnie jest to $k = 5$ lub $k = 10$. Następnie k -krotnie powtarzany jest proces wyboru jednego i -tego zbioru, gdzie i oznacza kolejne liczby z przedziału $<1; k>$. Zbiór o numerze i wybierany jest jako

zbiór walidacyjny, natomiast reszta $k - 1$ zbiorów służą jako dane uczące. Należy pamiętać, żeby w każdym przebiegu uczyć nową instancję modelu. W rezultacie model uczony jest k -krotnie na różnych kombinacjach danych. Jako wynik końcowy bierze się pod uwagę średnią skuteczność wszystkich modeli. Sposób podziału danych w tej metodzie ilustruje rysunek 5.5.



Rys. 5.5. Podział danych według metody k-krotnej walidacji krzyżowej

Przedstawioną metodę można rozszerzyć do jej iteracyjnej wersji polegającej na tym, że cały proces k -krotnej walidacji krzyżowej wykonywany jest n razy. Dodatkowo po każdej iteracji można zastosować mieszanie danych. Walidacja tą metodą co prawda trwa o wiele dłużej, gdyż model walidowany jest $k * n$ razy, jednak pozwala uzyskać bardziej wiarygodne wyniki, gdy nie jest dostępna zbyt duża liczba danych.

5.2.3. Implementacja walidacji

Zaimplementowane zostały obie metody podziału danych, jednak ze względu na swoje zalety metody iteracyjnej k -krotnej walidacji krzyżowej to właśnie ona została użyta do walidacji w procesie uczenia. Podział danych według obu metod został wykonany za pomocą biblioteki *sklearn*, która dostarcza przygotowane do tego celu funkcje (patrz listing 5.5).

Tak przygotowane dane podawane są do modelu w n iteracjach za pomocą prostej pętli. Wyniki z każdej iteracji zapisywane są do tablicy, a na koniec obliczana jest ich średnia. Proces iteracyjnego uruchamiania uczenia wraz z obliczaniem wyników zaprezentowany został na listingu 5.6

Listing 5.6. Iteracyjne wywołanie procesu uczenia

```

1 number_of_iterations = 5
2
3 avg_accuracies = []
4 std_accuracies = []
5
6 for iteration in range(0, number_of_iterations):
7     iteration_number = iteration + 1
8     print("Iteration", iteration_number)
```


Listing 5.5. Podział danych na podzbiory treningowe i testowe

```

1 from sklearn.model_selection import train_test_split, StratifiedKFold
2 from data_reader import get_data
3
4 def load_data_kfold(folds_number, test_size=0.05):
5     x, y = get_data(CHUNK_SIZE_IN_SECONDS)
6
7     x_train, x_test, y_train, y_test = train_test_split(x,
8                                                         y,
9                                                         test_size=test_size)
10
11     folds = list(StratifiedKFold(n_splits=folds_number,
12                                 shuffle=True,
13                                 random_state=1).split(x_train, y_train))
14
15     return folds, x_train, y_train, x_test, y_test
16
17 def load_data_train_test(test_size=0.05):
18     x, y = get_data(CHUNK_SIZE_IN_SECONDS)
19
20     x_train, x_test, y_train, y_test = train_test_split(x,
21                                                         y,
22                                                         test_size=test_size)
23
24     return x_train, y_train, x_test, y_test

```

```

9
10 score, best_model_score = run_pipeline(create_model=model,
11                                       folds=folds,
12                                       x=x_train,
13                                       y=y_train,
14                                       epochs=100)
15
16 accuracy = [row[1] for row in best_model_score]
17
18 avg_accuracy = np.mean(accuracy)
19 print("Best models average validation accuracy: {}".format(round(avg_accuracy, 6)))
20
21 avg accuracies.append(avg_accuracy)
22
23 grand_mean_avg = np.mean(avg accuracies)
24 print("~~~Grand mean of average accuracy: {}".format(round(grand_mean_avg, 6)))

```

Do uruchamiania procesu uczenia używana jest funkcja *run_pipeline*, która szczegółowo opisana zostanie w rozdziale 5.3.

5.3. Uruchomienie uczenia właściwego

Do uruchomienia procesu uczenia właściwego służy funkcja *run_pipeline()*, która wykonuje następujące czynności:

- tworzy model na podstawie dostarczonych specyfikacji (5.3.1),
- pobiera odpowiednie porcje danych dla każdego kroku z przygotowanych wcześniej zbiorów (5.2.3),
- tworzy listę tzw. *callback*’ów (5.3.2),
- uruchamia na modelu metodę rozpoczynającą uczenie (5.3.3),
- wczytuje wagi najlepszego modelu i sprawdza jego skuteczność (5.3.4).

Kod metody *run_pipeline* zaprezentowany został na listingu 5.7. Poszczególne kroki zostaną opisane w kolejnych rozdziałach.

Listing 5.7. Implementacja metody uruchamiającej uczenie właściwe

```

1 def run_pipeline(create_model, folds, x, y, epochs):
2     best_model_score = []
3
4     for fold_number, (train_idx, val_idx) in enumerate(folds):
5         print('\nFold: ', fold_number)
6         # wybór odpowiednich podzbiorów
7         x_train_cv = x[train_idx]
8         y_train_cv = y[train_idx]
9         x_valid_cv = x[val_idx]
10        y_valid_cv = y[val_idx]
11
12        input_shape = x.shape[1:]
13
14        # stworzenie modelu
15        model, model_description = create_model(input_shape)
16
17        # utworzenie callback'ów
18        callbacks = callbacks_list("{} Fold: {}".format(model_description,
19                                                         fold_number))
20
21        # rozpoczęcie uczenia
22        history = model.fit(x_train_cv,
23                           y_train_cv,
24                           epochs=epochs,
25                           batch_size=16,
26                           callbacks=callbacks,
27                           validation_data=(x_valid_cv, y_valid_cv),
28                           verbose=0)
29
30        # wczytanie wag najlepszego modelu i sprawdzenie jego skuteczności
31        model.load_weights("tmp/best_model.h5")
32        best_model_score.append(model.evaluate(x_valid_cv, y_valid_cv, batch_size=16,
33                                              verbose=0))
34        print("--Best model validation accuracy: %.2f%%" %
35              (best_model_score[fold_number][1]*100))
36
37    return best_model_score

```

5.3.1. Budowa modelu

Model tworzony jest na podstawie dostarczonych specyfikacji w postaci funkcji. Funkcja wykonuje kolejne kroki budujące model z kolejnych warstw, które dostępne są jako gotowe komponenty z biblioteki Keras. Każda z warstw jest parametryzowana wybranymi wartościami. Po utworzeniu modelu jest on kompilowany. Na tym etapie podawane są również informacje odnośnie tego jaki optymalizator, funkcja strat oraz metryka skuteczności ma zostać użyta. Dodatkowo tworzony jest również opis modelu wykorzystywany później w procesie monitorowania.

Na listingu 5.8 została zaprezentowana funkcja tworząca przykładowy model splotowej sieci neuronowej z 1-wymiarowym filtrem.

Listing 5.8. Tworzenie przykładowego modelu

```

1 def conv_1D_with_adam(input_shape):
2     # opis modelu tworzony na podstawie nazwy funkcji
3     description = get_function_name()
4
5     # specyfikacja modelu tworzonego sekwencyjnie
6     model = Sequential()
7
8     # dodanie warstwy splotowej z 32 1-wymiarowymi filtrami o rozmiarze 6 z funkcją
9     # aktywacji 'relu'

```

```

9      model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu',
10                     input_shape=input_shape))
11      # warstwa max pooling'u o rozmiarze 2
12      model.add(MaxPooling1D(pool_size=2))
13      # warstwa zmniejszająca liczbę wymiarów danych
14      model.add(Flatten())
15      # warstwa typu 'fully-connected' o rozmiarze 64 neuronów
16      model.add(Dense(64, activation='relu'))
17      # warstwa typu 'fully-connected' z jednym neuronem i sigmoidalną funkcją aktywacji,
18      # która na wyjściu zwróci wartość '0' lub '1'
19      model.add(Dense(1, activation='sigmoid'))
20      # kompilacja modelu z użyciem optymalizatora Adam, funkcji strat binarnej entropii
21      # krzyżowej oraz dokładności (accuracy) jako metryki
22      model.compile(optimizer=Adam(),
23                   loss='binary_crossentropy',
24                   metrics=['acc'])
25      return model, description

```

5.3.2. Lista callback'ów

Dodatkowym parametrem przekazywanym do procesu uczenia jest lista tzw. *callback*'ów, czyli wywołań zwrotnych, które umożliwiają otrzymywanie informacji z wnętrza modelu podczas jego nauki.

Zostały utworzone 4 następujące callbacki:

- **EarlyStopping** - pozwala na wcześniejsze zatrzymanie procesu uczenia w przypadku, gdy monitorowana metryka nie została poprawiona przez określoną ilość epok,
- **LearningRateScheduler** - implementuje adaptacyjną metodę zmiany współczynnika uczenia według podanych reguł,
- **ModelCheckpoint** - umożliwia zapisanie modelu, który posiada najlepsze dopasowanie według określonej metryki,
- **TensorBoard** - tworzy logi z procesu uczenia w podanym folderze, które mogą być użyte do monitorowania przez narzędzie TensorBoard 5.4.

Listing 5.9 przedstawia implementację metod tworzących *callback*'i.

Listing 5.9. Tworzenie listy *callback*'ów

```

1  def step_decay(epoch):
2      initial_lrate=0.1
3      drop=0.6
4      epochs_drop = 10.0
5      lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
6
7      return lrate
8
9
10 def callbacks_list(description):
11     return [
12         callbacks.EarlyStopping(
13             monitor='val_acc',
14             patience=30
15         ),
16         callbacks.LearningRateScheduler(step_decay),
17     ]

```

```

18     callbacks.ModelCheckpoint(
19         filepath='tmp/best_model.h5',
20         monitor='val_acc',
21         save_best_only=True
22     ),
23     callbacks.TensorBoard(
24         log_dir='tmp/logs/{}'.format(description, create_current_time()),
25         histogram_freq=0,
26         write_graph=True,
27         write_images=True
28     )
29 ]

```

5.3.3. Funkcja dopasowania

Wykonywana na modelu funkcja *fit()* rozpoczyna procedurę uczenia, czyli dopasowania do danych. Jako argumenty przykazywane są do niej dane treningowe i walidacyjne, liczba epok uczących, *callback*'i oraz inne parametry. Podczas nauki wyświetlane są informacje o postępie uczenia oraz zwracany jest raport zawierający historię wartości metryk w kolejnych epokach, na podstawie których można np. narysować wykres przedstawiający postępy. Te wartości nie będą jednak używane, gdyż cała historia i postępy uczenia są ładowane do programu monitorującego TensorBoard z logów tworzonych przez callback TensorBoard (patrz rozdział 5.4). Dodatkowo dla każdej iteracji zapisywany jest najlepszy model, który na sam koniec zostanie załadowany i sprawdzony pod kątem skuteczności (patrz listing 5.3.4).

Listing 5.10. Wywołanie metody *fit()* na modelu

```

1 history = model.fit(x_train_cv,
2                     y_train_cv,
3                     epochs=epochs,
4                     batch_size=16,
5                     callbacks=callbacks,
6                     validation_data=(x_valid_cv, y_valid_cv),
7                     verbose=0)

```

5.3.4. Sprawdzenie skuteczności

W celu sprawdzenia skuteczności wczytywany jest najlepiej dopasowany model z danej iteracji i jest on sprawdzany metodą *evaluate()* na danych walidacyjnych. Wynik jest następnie wyświetlany i zapisywany do tablicy, która posłuży do policzenia średniej skuteczności modelu. Przedstawione działania wykonuje kod zaprezentowany na listingu 5.11

Listing 5.11. Sprawdzenie skuteczności najlepszego modelu

```

1 model.load_weights("tmp/best_model.h5")
2 best_model_score.append(model.evaluate(x_valid_cv, y_valid_cv, batch_size=16,
3                                       verbose=0))
4
5 print("--Best model validation accuracy: %.2f%%" %
6       (best_model_score[fold_number][1]*100))

```

5.4. Monitorowanie

Model sieci neuronowej po zakończeniu procesu nauki zwraca obiekt zawierający raport z wynikami. W sytuacji, gdy proces uczenia przeprowadzany jest kilkakrotnie należałoby ręcznie zarządzać obiektami raportów. Dodatkowo, przedstawienie wyników w sposób czytelny i łatwy do zinterpretowania wymaga zaimplementowania metod wizualizacyjnych np. rysowanie wykresu. Nie ma jednak potrzeby ręcznej implementacji wyżej wymienionych funkcjonalności, gdyż istnieją dedykowane narzędzia umożliwiające prezentację wyników w czasie rzeczywistym w przyjaznej dla użytkownika formie.

Do monitorowania procesu uczenia zostało użyte narzędzie *TensorBoard*, które dostarczone jest razem z biblioteką *TensorFlow*. Pozwala ono m.in. na graficzną wizualizację postępów w procesie uczenia oraz automatyczne generowanie schematu modelu sieci neuronowej. Jest to narzędzie uruchamiane w przeglądarce, które wykorzystuje logi wygenerowane przez *callback.TensorBoard* omawiany w rozdziale 5.3.2.

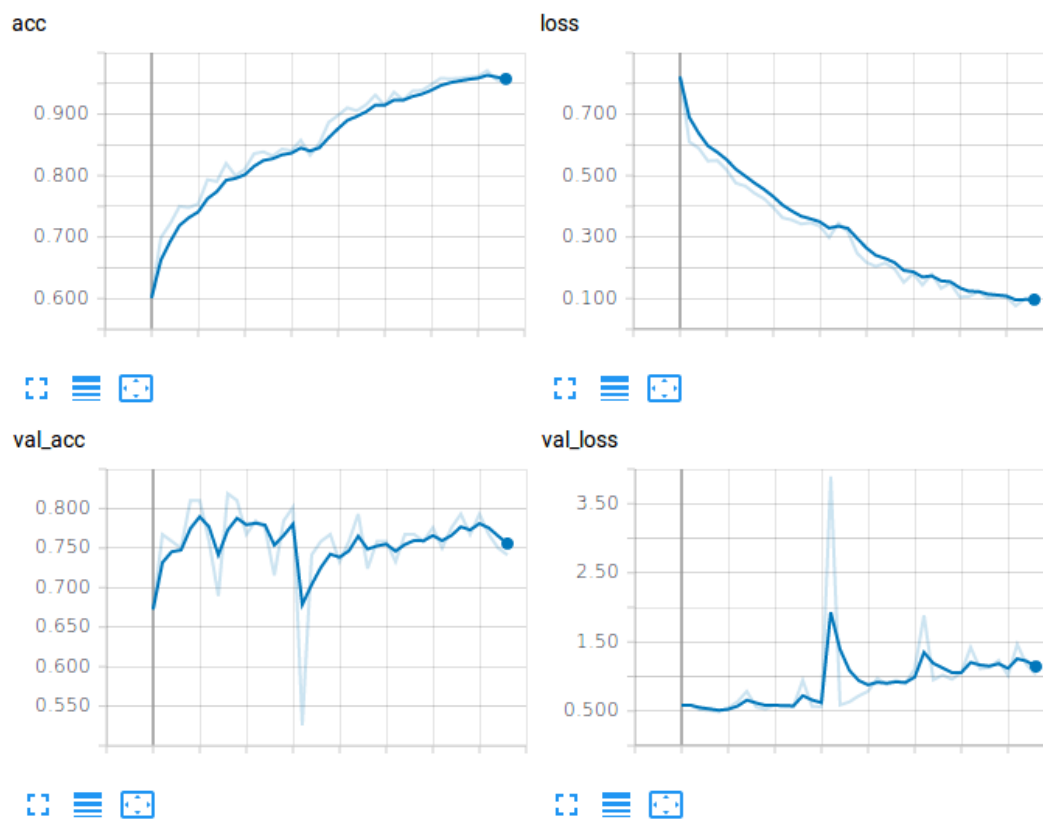
Na podstawowym widoku przedstawione są 4 wykresy ilustrujące następujące wartości:

- **acc** - dokładność w procesie uczenia,
- **loss** - wartość funkcji strat w procesie uczenia,
- **val_acc** - dokładność w procesie walidacji,
- **val_loss** - wartość funkcji strat w procesie walidacji.

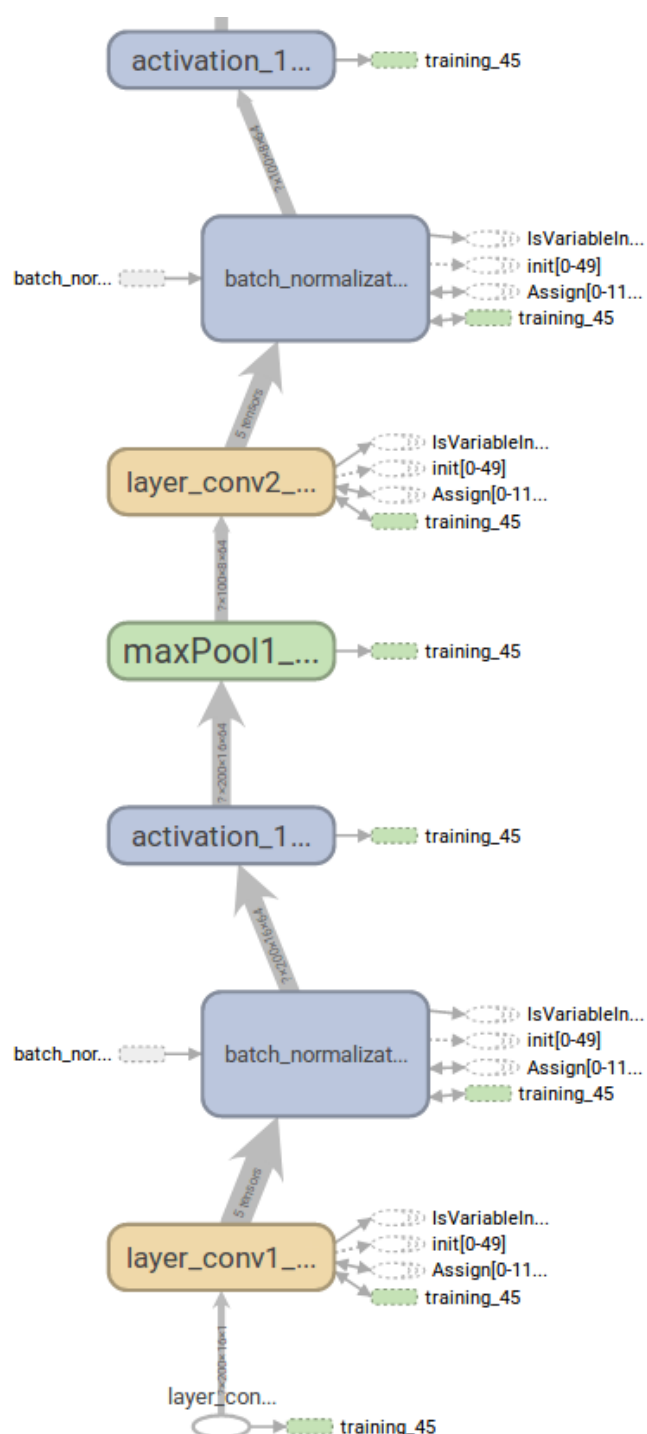
Wykresy zostały zaprezentowane na rysunku 5.6. Są one interaktywne, więc można sprawdzić dokładną wartość w każdym punkcie wykresu.

Wizualna reprezentacja utworzonego modelu również może być pomocna przy ocenie architektury tworzonej sieci neuronowej. Fragment przykładowego schematu wygenerowany przez *TensorBoard* na podstawie modelu utworzonego w kodzie pokazany został na rysunku 5.7.

TensorBoard udostępnia również wiele innych funkcjonalności, które nie zostały użyte jak np. rysowanie wykresów własnoręcznie zdefiniowanych metryk, monitorowanie i wizualizację rozkładu danych w grafie itp. Więcej informacji na temat możliwości *TensorBoard* można znaleźć w dokumentacji [12].



Rys. 5.6. Dynamicznie tworzone wykresy w programie TensorBoard



Rys. 5.7. Fragment schematu modelu wygenerowany przez TensorBoard

5.5. Wybór rodzaju sieci neuronowej

Wybór rodzaju oraz wstępnej architektury sieci neuronowej ma kluczowe znaczenie. To właśnie te czynniki będą w głównej mierze decydować o skuteczności modelu. Po dobraniu rodzaju sieci neuronowej, który umożliwia uzyskanie jak najlepszych wyników następuje faza optymalizacji, czyli dostosowywania parametrów, która zostanie opisana w rozdziale 5.6.

Przy wyborze modelu warto zacząć od rozwiązań najprostszych. Proste modele będą potrzebowały zdecydowanie mniej czasu na naukę, niż te złożone, dlatego dzięki użyciu tego podejścia będzie można w krótkim czasie uzyskać pierwsze rezultaty. Może się również okazać, że bardzo prosty model jest dostatecznie skuteczny w aktualnie rozpatrywanym problemie.

Podczas próby znalezienia odpowiedniego rodzaju sieci neuronowej rozpatrzone zostały następujące opcje:

- klasyczna sieć typu *fully-connected*,
- sieć splotowa z filtrem 1-wymiarowym,
- sieć splotowa z filtrem 2-wymiarowym,
- sieć rekurencyjna,
- połączenie sieci splotowej z rekurencyjną.

5.5.1. Klasyczna sieć neuronowa typu fully connected

Pierwszym, najprostszym rozwiązaniem, które zostało przetestowane jest klasyczna sieć neuronowa typu *fully connected*. W bibliotece Keras warstwy tworzące taką sieć noszą nazwę *Dense*. Model początkowy, który został stworzony zaprezentowano na listingu 5.12.

Listing 5.12. Model sieci neuronowej typu *fully connected*

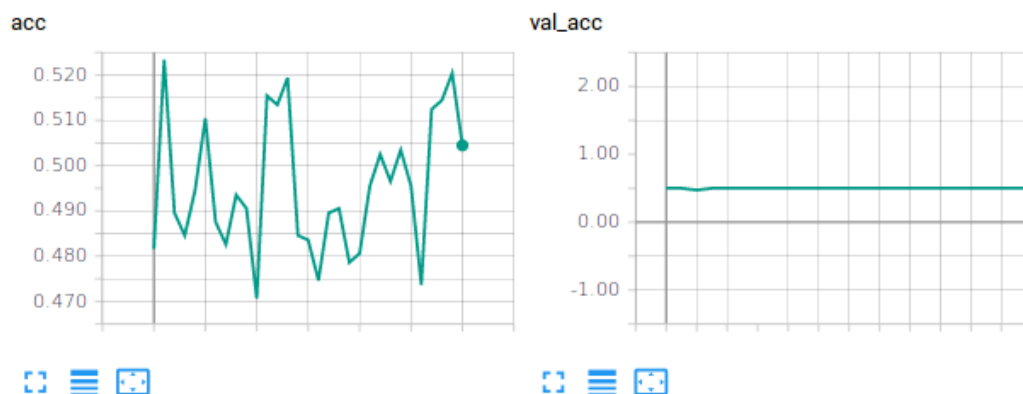
```
1 model = Sequential()
2 model.add(Flatten(input_shape=input_shape))
3 model.add(Dense(1000, kernel_initializer='normal', activation='relu'))
4 model.add(Dense(30, kernel_initializer='normal', activation='relu'))
5 model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
6
7 sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
8 model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
```

Sieć składa się z warstwy wejściowej *Flatten*, która odpowiada za zmniejszenie stopnia wymiarowości danych tak, aby mogły być użyte w sieci tego typu. Przez tę operację traczone są pewne informacje o sąsiedztwie danych, gdyż teraz są one jedynie wektorem zamiast wielowymiarową tablicą, jednak jest ona wymagana. Kolejne warstwy stanowią warstwy *Dense* zawierające różną ilość neuronów. Początkowo zostały użyte wartości 1000 i 30. Sieć zakończona jest warstwą z jednym neuronem i sigmoidalną funkcją aktywacji, która zwraca na wyjściu wartość 1 lub 0. Jako optymalizator użyty został *SGD* (*ang. stochastic gradient descent*) z funkcją strat binarnej entropii krzyżowej (*ang. binary crossentropy*). Podsumowanie modelu wygenerowane przez bibliotekę *Keras*, opisujące ilość warstw oraz ich wielkości, zamieszczono na listingu 5.13.

Listing 5.13. Podsumowanie modelu sieci neuronowej typu *fully connected*

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 3200)	0
dense_3 (Dense)	(None, 1000)	3201000
dense_4 (Dense)	(None, 30)	30030
dense_5 (Dense)	(None, 1)	31
Total params: 3,231,061		
Trainable params: 3,231,061		
Non-trainable params: 0		

Proces nauki zaproponowanej sieci neuronowej przebiegał szybko ze względu na jej prostotę. Jedna pełna iteracja trwała jedynie około 90 sekund. Wyniki były jednak bardzo słabe. Skuteczność podczas uczenia modelu oscylowała wokół wartości 50% (48% - 52%). Skuteczność walidacyjna również była praktycznie stała i zawsze bliska 50% 5.8.

**Rys. 5.8.** Wyniki skuteczności sieci typu *fully-connected*

Można więc zauważyć, że sieć działała praktycznie w sposób losowy, gdyż tyle wynosi właśnie prawdopodobieństwo przewidzenia wyniku z pośród dwóch wartości. Pomimo próby modyfikacji sieci poprzez zwiększenie/zmniejszenie liczby warstw, zwiększenie/zmniejszenie liczby neuronów, zmiany optymalizatora i funkcji strat, wciąż przynosiła ona podobne wyniki. Nie udało się osiągnąć lepszych rezultatów dla tego typu sieci neuronowej, dlatego nie była ona więcej wykorzystywana. Okazała się niewystarczająca do rozwiązania rozpatrywanego problemu.

5.5.2. Sieć splotowa z filtrem 1-wymiarowym

Kolejnym testowanym typem była sieć splotowa z 1-wymiarowym filtrem, której zastosowanie polecane jest do danych w formie szeregów czasowych np. dane z żyroskopu czy akcelerometru. W bibliotece Keras warstwy tego typu nazwane są *Conv1D* (ang. *Convolutional 1-dimension*). Filtrowymiarowy jest w stanie odnajdywać podobieństwa między danymi jedynie w jednym wymiarze. W przypadku dostępnych danych z EEG mogą być to wartości tylko z jednego kanału jednocze-

śnie. Oznacza to, że być może sieć tego typu nie będzie w stanie zauważyć zależności pomiędzy danymi znajdującymi się na różnych kanałach.

Wstępny model sieci, który został wybrany przedstawiony został na listingu 5.14. Składa się on z dwóch warstw konwolucyjnych o odpowiednio 32 i 16 filtrach z warstwami *max pooling*'u. Sieci splotowe zwykle zakończone są kilkoma warstwami typu *fully-connected*. W tym wypadku została zastosowana jedna warstwa z 64 neuronami oraz warstwa wyjściowa z jednym neuronem i sigmoidalną funkcją aktywacji. Warstwa końcowa tego typu będzie wykorzystywana w każdym modelu, gdyż daje ona możliwość zwrócenia na wyjściu z sieci wartość 0 lub 1, które oznaczają brak lub pojawienie się ataku. Podsumowanie modelu znajduje się na listingu 5.15

Listing 5.14. Model splotowej sieci neuronowej z 1-wymiarowym filtrem

```

1 model = Sequential()
2
3 model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu',
4   input_shape=input_shape))
5 model.add(MaxPooling1D(pool_size=2))
6
7 model.add(Conv1D(filters=16, kernel_size=6, padding='same', activation='relu'))
8 model.add(MaxPooling1D(pool_size=2))
9
10 model.add(Flatten())
11 model.add(Dense(64, activation='relu'))
12 model.add(Dense(1, activation='sigmoid'))
13
14 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])

```

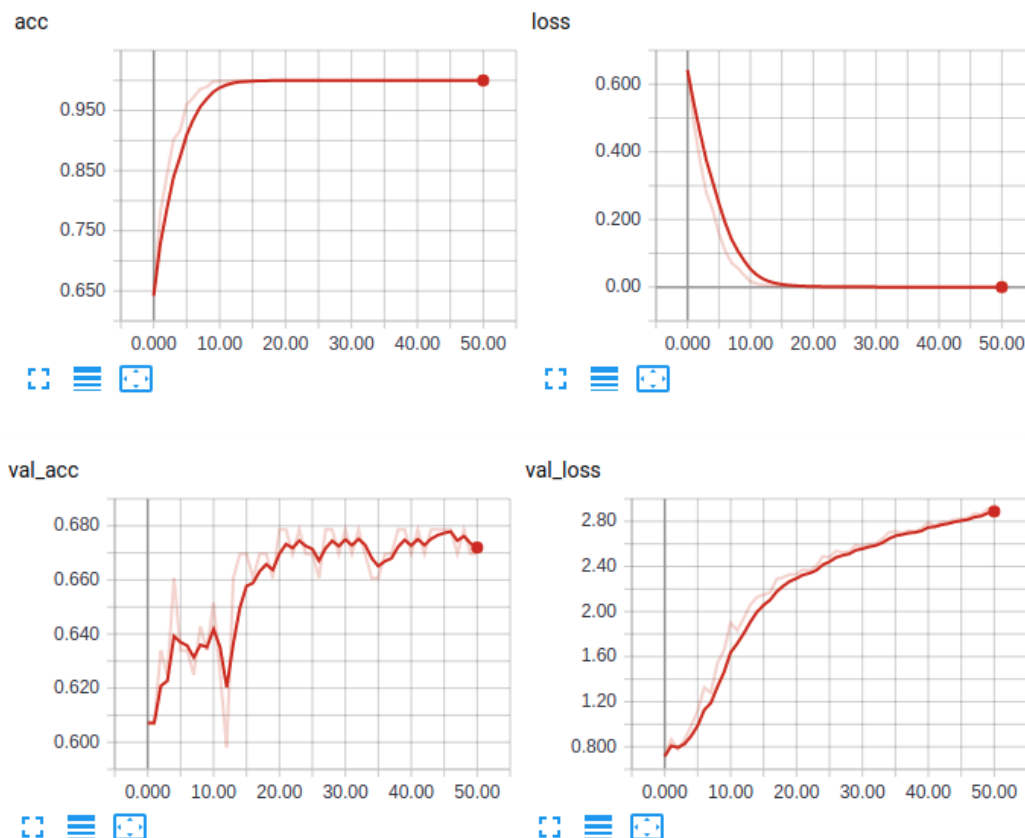
Listing 5.15. Podsumowanie modelu splotowej sieci neuronowej z 1-wymiarowym filtrem

```

1 -----
2 Layer (type)                Output Shape                Param #
3 -----
4 conv1d_2 (Conv1D)           (None, 200, 32)            3104
5 -----
6 dropout_5 (Dropout)         (None, 200, 32)            0
7 -----
8 max_pooling1d_1 (MaxPooling1 (None, 100, 32)            0
9 -----
10 conv1d_3 (Conv1D)           (None, 100, 16)            3088
11 -----
12 dropout_6 (Dropout)         (None, 100, 16)            0
13 -----
14 max_pooling1d_2 (MaxPooling1 (None, 50, 16)            0
15 -----
16 flatten_5 (Flatten)         (None, 800)                0
17 -----
18 dense_9 (Dense)             (None, 64)                 51264
19 -----
20 dense_10 (Dense)            (None, 1)                  65
21 =====
22 Total params: 57,521
23 Trainable params: 57,521
24 Non-trainable params: 0
25 -----

```

Skonstruowana w ten sposób sieć neuronowa uczyła się co prawda dłużej, niż sieć typu *fully-connected*, gdyż czas nauki wynosił około 240 sekund, jednak wyniki były o wiele lepsze i oscylowały w okolicach **70%**. Jest to bardzo dobry wynik w porównaniu do poprzedniego biorąc pod uwagę prostą architekturę sieci. Sieć jednak bardzo szybko się przeuczała (*ang. overfitting*). Można to zauważyć po osiągnięciu 100% skuteczności podczas procesu nauki (*acc*) oraz wzroście wartości funkcji strat podczas walidacji (*val_loss*) (patrz rysunek 5.9).



Rys. 5.9. Wyniki prostej sieci spłotowej z 1-wymiarowym filtrem

Wyniki były obiecujące, więc sieć tego typu została wzięta pod uwagę w dalszych pracach polegających na optymalizacji w celu uzyskania jak najlepszych wyników (rozdział 5.6).

5.5.3. Sieć spłotowa z filtrem 2-wymiarowym

Następnym typem sieci, która została przetestowana była sieć spłotowa z filtrem 2-wymiarowym (*Conv2D*). Sieć ta jest bardzo podobna do konwolucyjnej sieci jednowymiarowej opisywanej w poprzednim podrozdziale 5.5.2. Różnicą jest typ zastosowanego filtra, który w tym przypadku jest 2-wymiarowy. Pozwala więc analizować dane w dwuwymiarowej przestrzeni. Sieci tego typu używane są przeważnie do rozpoznawania obrazów, jednak równie dobrze mogą się sprawdzić też na danych innego typu. W przypadku EEG sieć z 2-wymiarowym filtrem będzie w stanie analizować szereg czasowy obejmując kilka kanałów jednocześnie, dzięki temu możliwe będzie zauważenie powiązania pomiędzy danymi z innych kanałów.

Analogicznie do modelu sieci z filtrem jednowymiarowym został stworzony podobny z 2-wymiarowym filtrem (patrz listing 5.16). Podsumowanie modelu zaprezentowano na listingu 5.17.

Nauka tego modelu sieci trwała trochę dłużej, niż poprzedniego, gdyż czas wynosił około 300 sekund. Okazało się jednak, że sieć która różni się od poprzedniej jedynie wymiarem filtra osiągnęła skuteczność na poziomie **75%**. Po wynikach zapre-

Listing 5.16. Model spłotowej sieci neuronowej z 2-wymiarowym filtrem

```

1 model = Sequential()
2
3 model.add(Conv2D(32,(3,3),strides = (1,1),padding='same', activation = 'relu',
4               input_shape=input_shape))
5 model.add(MaxPooling2D((2,2)))
6
7 model.add(Conv2D(16,(3,3),strides = (1,1),name='conv3', activation = 'relu'))
8 model.add(MaxPooling2D((2,2)))
9
10 model.add(Flatten())
11 model.add(Dense(64,activation = 'relu'))
12 model.add(Dense(1, activation='sigmoid'))
13
14 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])

```

Listing 5.17. Podsumowanie modelu spłotowej sieci neuronowej z 2-wymiarowym filtrem

```

1 -----
2 Layer (type)                Output Shape                Param #
3 -----
4 conv2d_1 (Conv2D)           (None, 200, 16, 32)        320
5 -----
6 max_pooling2d_1 (MaxPooling2 (None, 100, 8, 32)        0
7 -----
8 conv3 (Conv2D)              (None, 98, 6, 16)          4624
9 -----
10 max_pooling2d_2 (MaxPooling2 (None, 49, 3, 16)          0
11 -----
12 flatten_1 (Flatten)         (None, 2352)                0
13 -----
14 dense_1 (Dense)             (None, 64)                  150592
15 -----
16 dense_2 (Dense)             (None, 1)                   65
17 -----
18 Total params: 155,601
19 Trainable params: 155,601
20 Non-trainable params: 0
21 -----

```

zentowanych na rysunku 5.10 można jednak zauważyć, że pomimo lepszej skuteczności sieć tego typu również bardzo szybko ulega przeuczeniu. Problem ten będzie rozwiązywany podczas próby jej optymalizacji w rozdziale 5.6.

5.5.4. Sieć rekurencyjna

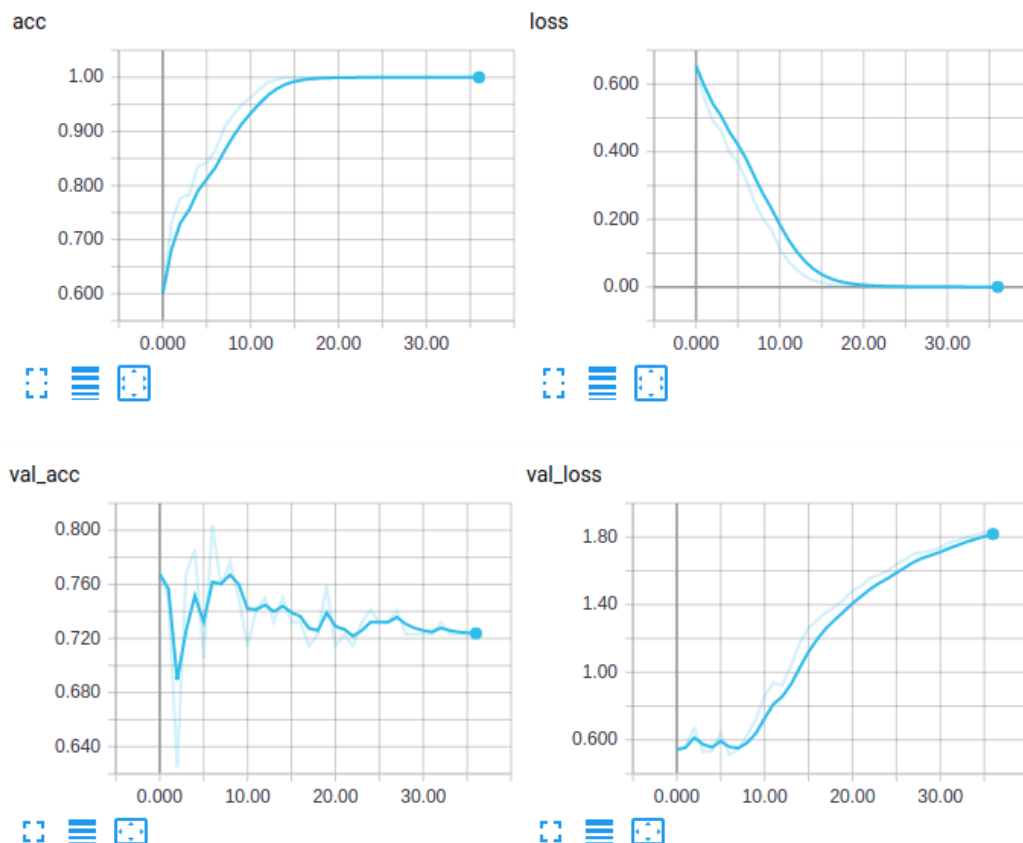
W następnym kroku przetestowana została rekurencyjna sieć neuronowa. Początkowy model sieci, który został utworzony zaprezentowany został na listingu 5.18. Podsumowanie modelu zamieszczono na listingu 5.19.

Listing 5.18. Model rekurencyjnej sieci neuronowej

```

1 model = Sequential()
2
3 model.add(LSTM(100, input_shape=input_shape))
4 model.add(Dense(1, activation='sigmoid'))
5
6 model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

```



Rys. 5.10. Wyniki prostej sieci spłotowej z 2-wymiarowym filtrem

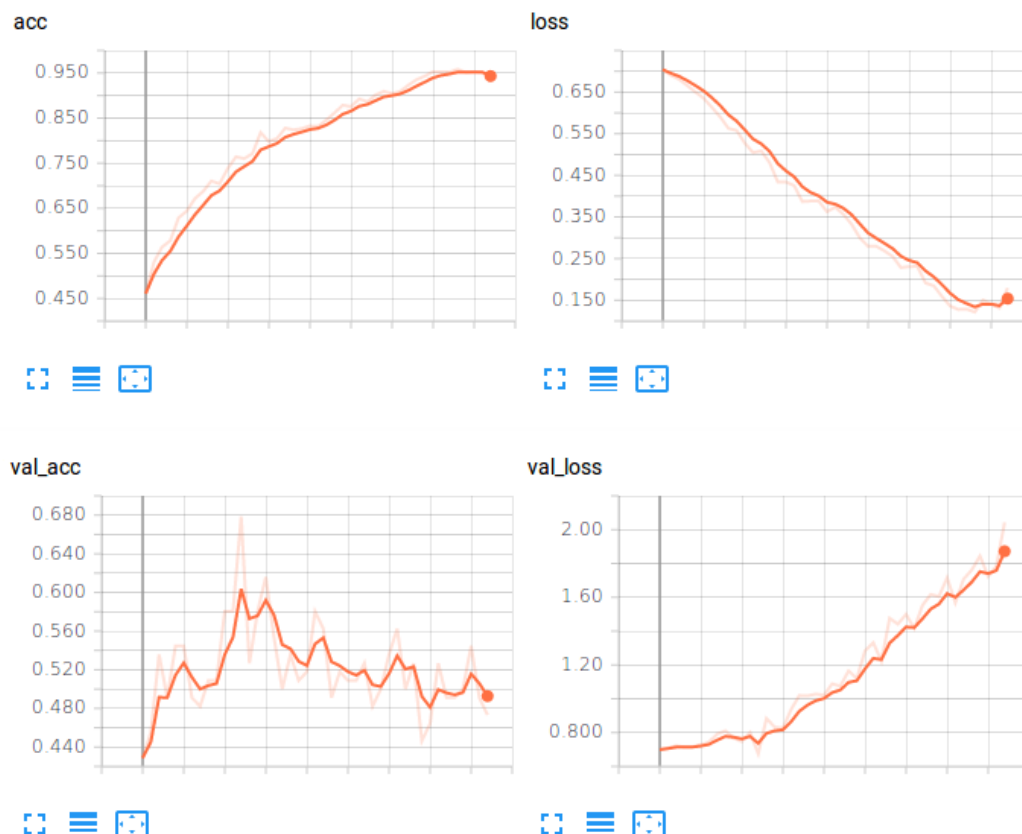
Listing 5.19. Podsumowanie modelu rekurencyjnej sieci neuronowej

1	-----		
2	Layer (type)	Output Shape	Param #
3	-----	-----	-----
4	lstm_2 (LSTM)	(None, 100)	46800
5	-----	-----	-----
6	dense_17 (Dense)	(None, 1)	101
7	-----	-----	-----
8	Total params: 46,901		
9	Trainable params: 46,901		
10	Non-trainable params: 0		
11	-----		

Składa się ona z jednej warstwy typu *LSTM* (ang. *Long Short-Term Memory*). Sieć mimo prostej budowy uczy się bardzo długo. Czas pełnej iteracji nauki wynosi aż 5700 sekund. Wynika to ze złożoności sposobu funkcjonowania tego typu sieci. Pomimo dłuższego czasu nauki zaproponowana sieć osiąga skuteczność na poziomie jedynie ok. **60%** 5.11.

Jest to wynik o wiele gorszy w porównaniu do sieci konwolucyjnych testowanych w poprzednich rozdziałach. Próby rozbudowy i prostej optymalizacji sieci nie przynosiły pożądanych skutków, dlatego została ona pominięta w dalszych badaniach.

Poczyniono jednak próby połączenia tego typu sieci z sieciami konwolucyjnymi. Wyniki przeprowadzonych prób opisane zostały w rozdziale 5.5.5



Rys. 5.11. Wyniki prostej sieci rekurencyjnej

5.5.5. Połączenie sieci splotowej z rekurencyjną

Ostatnim typem sieci, którego skuteczność została sprawdzona jest połączenie sieci splotowej z rekurencyjną tzw. *CNN LSTM*. Jest to bardziej rozbudowana struktura sieci polegająca na użyciu kilku warstw konwolucyjnych zakończonych warstwami rekurencyjnymi. Prosty model sieci zbudowany według tej koncepcji zaprezentowany został na listingu 5.20. Jego podsumowanie zamieszczono na listingu 5.21.

Listing 5.20. Model sieci typu CNN LSTM

```

1 model = Sequential()
2
3 model.add(Conv1D(filters=64, kernel_size=6, padding='same', activation='relu',
4               input_shape=input_shape))
5 model.add(MaxPooling1D(pool_size=2))
6 model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu'))
7 model.add(MaxPooling1D(pool_size=2))
8 model.add(LSTM(100))
9
10 model.add(Dense(128, activation='relu'))
11 model.add(Dense(1, activation='sigmoid'))
12
13 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])

```

Pomimo bardziej złożonej struktury proces nauki utworzonej sieci przebiegał krócej, niż w przypadku tej składającej się jedynie z warstwy rekurencyjnej i wyniósł

Listing 5.21. Podsumowanie modelu sieci typu CNN LSTM

Layer (type)	Output Shape	Param #
conv1d_2 (Conv1D)	(None, 200, 32)	3104
dropout_1 (Dropout)	(None, 200, 32)	0
max_pooling1d_1 (MaxPooling1D)	(None, 100, 32)	0
conv1d_3 (Conv1D)	(None, 100, 16)	3088
dropout_2 (Dropout)	(None, 100, 16)	0
max_pooling1d_2 (MaxPooling1D)	(None, 50, 16)	0
lstm_4 (LSTM)	(None, 100)	46800
dense_19 (Dense)	(None, 128)	12928
dense_20 (Dense)	(None, 1)	129
Total params: 66,049		
Trainable params: 66,049		
Non-trainable params: 0		

około 1800 sekund. Spowodowane jest to tym, że dane wejściowe ulegają zmniejszeniu za pomocą warstw konwolucyjnych i wstępnie przetworzone trafiają na warstwę rekurencyjną. Dzięki temu warstwa rekurencyjna potrzebuje mniej czasu do nauki, niż w przypadku, gdy podawane jej są nieprzygotowane dane.

Osiągnięte wyniki były co prawda lepsze od sieci rekurencyjnej, jednak nie przewyższały wyników modeli zbudowanych z użyciem warstw konwolucyjnych. Osiągnięte wyniki wahały się w granicach **65%** 5.12.

Zostały podjęte próby rozszerzenia modelu sieci oraz zmiany parametrów, jednak wyniki nie zostały w znaczącym stopniu poprawione. Ze względu na to sieć tego typu nie została użyta w dalszych badaniach.

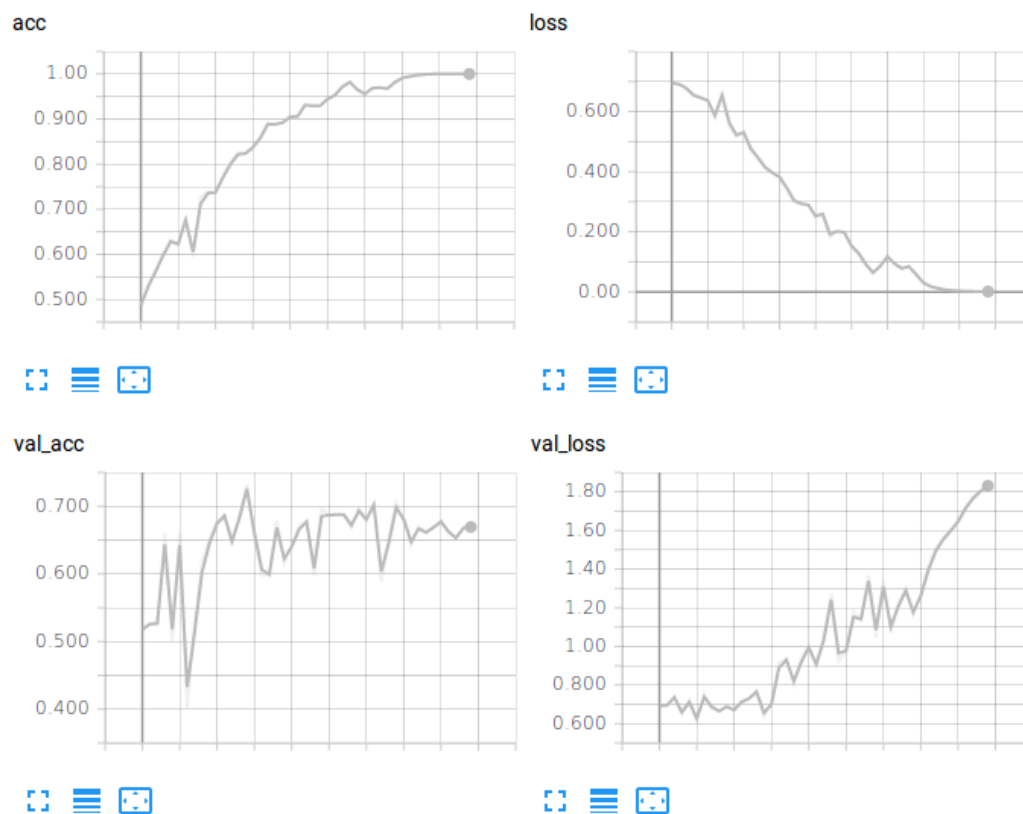
5.5.6. Podsumowanie

Podsumowanie wyników testowania skuteczności poszczególnych typów sieci zostały zaprezentowane w tabeli 5.1.

Tab. 5.1. Porównanie czasów nauki oraz skuteczności poszczególnych typów sieci

Nazwa	Średni czas jednej iteracji [s]	Średnia skuteczność [%]
Fully connected	90	50
CNN 1D	240	70
CNN 2D	300	75
LSTM	5700	60
CNN LSTM	1800	65

Można zauważyć, że najlepsze wyniki i stosunkowo nieduże czasy nauki osiągnęła sieć spłotowa z filtrem 2-wymiarowym. Model tej sieci został więc wybrany jako



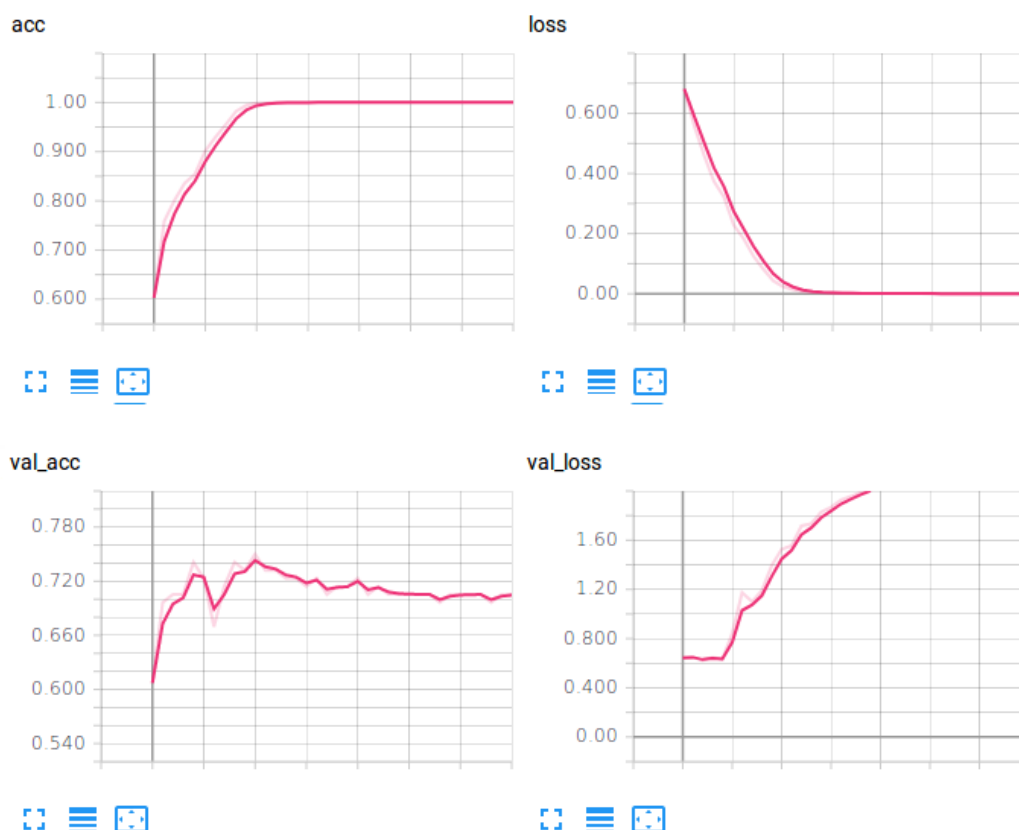
Rys. 5.12. Wyniki sieci konwolucyjnej połączonej z rekurencyjną

ten najlepiej przystosowany do rozwiązywania badanego problemu. Próba i sposoby optymalizacji wyników tego modelu zostały opisane w kolejnym rozdziale 5.6.

5.6. Optymalizacja

Sieć typu konwolucyjnego z filtrem 2-wymiarowym osiągała najlepsze wyniki wśród testowanych prototypów. To właśnie tej architekturze zostało więc poświęcone najwięcej pracy mającej na celu optymalizację i polepszenie otrzymywanych wyników. Wyniki już nawet przy zaprezentowanym prostym modelu były obiecujące, gdyż sieć osiągała średnio 75% skuteczności.

Największym i najczęściej występującym problemem podczas nauki sieci neuronowych jest przeuczenie (*ang. overfitting*), które polega na zbyt dużym dopasowaniu się modelu do danych uczących. Przeuczony model będzie posiadał zbyt małą umiejętność generalizacji, w wyniku czego będzie bardzo dobrze potrafił klasyfikować dane, które do tej pory widział, jednak nie poradzi sobie zbyt dobrze z zupełnie nowymi. Overfitting zwykle objawia się osiągnięciem skuteczności bliskiej 100% i wartości funkcji strat wynoszącej około 0 podczas procesu uczenia. Skutkiem tego są oczywiście o wiele gorsze wyniki podczas procesu walidacji. Tak jest również w przypadku tego prostego modelu, co można zaobserwować na wykresach prezentujących przebieg uczenia przedstawiony na rysunku 5.13.



Rys. 5.13. Wyniki modelu początkowej konwolucyjnej sieci neuronowej

Istnieje kilka sposobów przeciwdziałania nadmiernemu dopasowaniu. Zostaną one opisane w kolejnych rozdziałach.

5.6.1. Batch normalization

Pierwszym sposobem jest wykorzystanie *batch normalization*. Technika ta polega na normalizacji wartości wyjściowych z danej warstwy tak, aby miały one średnią 0 i odchylenie standardowe 1. Jest to zabieg podobny do tego przeprowadzonego podczas przygotowywania danych do procesu nauki z tą różnicą, że można go stosować dla wartości wyjściowych z warstw. Dodatkowo zmniejsza zależność wyników osiąganych przez sieć od wartości, którymi zainicjalizowane były wagi oraz poprawia przepływ gradientu. Więcej informacji dotyczącej *batch normalization* można znaleźć w publikacji poświęconej temu zagadnieniu [13].

W bibliotece *Keras* dostępne są gotowe warstwy o nazwie *BatchNormalization* dostarczające opisaną funkcjonalność. Wystarczy dodać warstwę poprzedzającą funkcję aktywacji 5.22.

Listing 5.22. Model sieci konwolucyjnej z warstwami BatchNormalization

```

1 model = Sequential()
2
3 model.add(Conv2D(32, (3,3), strides = (1,1), padding='same', input_shape=input_shape))
4 model.add(BatchNormalization())
5 model.add(Activation('relu'))
6 model.add(MaxPooling2D((2, 2)))
7
8 model.add(Conv2D(16, (3,3), strides = (1,1), padding='same'))
9 model.add(BatchNormalization())
10 model.add(Activation('relu'))
11 model.add(MaxPooling2D((2,2)))
12
13 model.add(Flatten())
14 model.add(Dense(64, activation = 'relu'))
15 model.add(Dense(1, activation='sigmoid'))
16
17 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])

```

Wykonanie tak prostej operacji pozwoliło na zwiększenie średniej skuteczności sieci do **77%**. Problem *overfitting*'u co prawda nie został całkowicie rozwiązany, jednak można zauważyć drobne postępy względem poprzedniej wersji modelu 5.14

5.6.2. Dropout

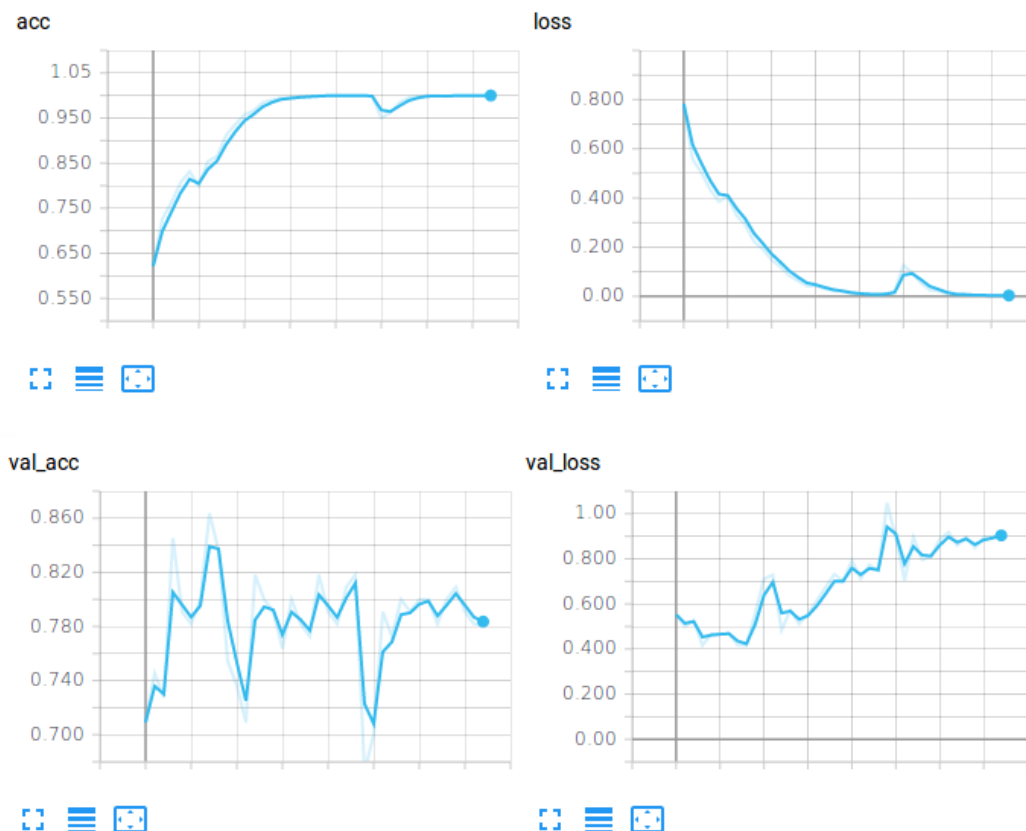
Kolejną techniką jest wykorzystanie tzw. *dropout*'u. Polega ona na tym, że wartości na warstwie wyjściowej są zerowane ze wskazanym prawdopodobieństwem. Dzięki temu podczas każdej epoki uczącej dane wyglądają trochę inaczej, niż poprzednio, dlatego sieć nie uczy się zawsze na podstawie tych samych danych, tylko ich drobnych modyfikacjach. W bibliotece *Keras* została zaimplementowana gotowa warstwa *Dropout*, która umożliwia wykorzystanie tej techniki 5.23. Została ona nałożona na wyniki warstwy typu *Dense*.

Listing 5.23. Model sieci konwolucyjnej z warstwami Dropout

```

1 model = Sequential()
2
3 model.add(Conv2D(32, (3,3), strides = (1,1), padding='same', input_shape=input_shape))
4 model.add(BatchNormalization())
5 model.add(Activation('relu'))
6 model.add(MaxPooling2D((2, 2)))
7
8 model.add(Conv2D(16, (3,3), strides = (1,1), padding='same'))
9 model.add(BatchNormalization())
10 model.add(Activation('relu'))

```



Rys. 5.14. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami *BatchNormalization*

```

11 model.add(MaxPooling2D((2,2)))
12
13 model.add(Flatten())
14 model.add(Dense(64, activation = 'relu'))
15 model.add(Dropout(0.25))
16 model.add(Dense(1, activation='sigmoid'))
17
18 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])

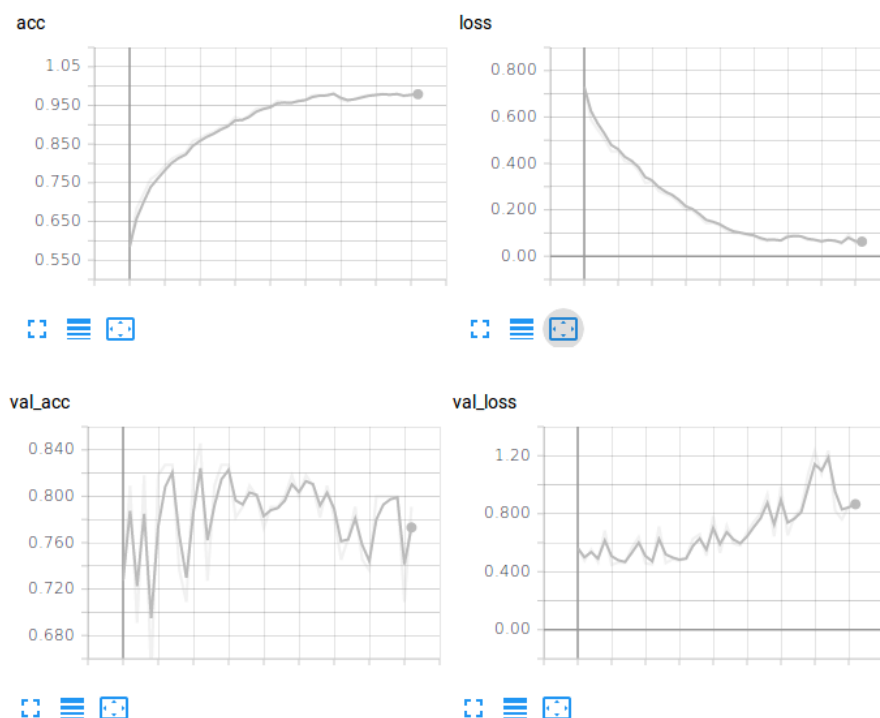
```

Po dodaniu *dropout*'u wyniki nieznacznie się poprawiły do wartości średnio **78%** oraz zmniejszony został problem przeuczenia. Podczas uczenia nie jest osiągnięta już skuteczność 100% oraz wartości funkcji strat równej 0. Podczas walidacji wartość funkcji strat dalej rośnie, jednak w mniejszym stopniu, niż poprzednio 5.15.

5.6.3. Wybór optymalizatora

Kolejnym czynnikiem wpływającym na skuteczność sieci jest wybrany optymalizator. Jego zadaniem jest modyfikacja wag na podstawie wartości funkcji strat. W bibliotece *Keras* istnieje wiele gotowych optymalizatorów, które mogą zostać wybrane podczas kompilacji modelu. Najbardziej popularnymi są: *Adam*, *RMSProp* oraz *SGD* (*ang. Stochastic gradient descent*). Wyniki osiągnięte przy użyciu poszczególnych optymalizatorów zamieszczone zostały w tabeli 5.2.

Optymalizatorem, który osiągał najlepsze wyniki był SGD. Dodatkowo opty-



Rys. 5.15. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami *Dropout*

Tab. 5.2. Porównanie skuteczności sieci przy użyciu poszczególnych optymalizatorów

Optymalizator	Średnia skuteczność [%]
Adam	78
SGD	79
RMSProp	76,5

malizator tego typu bardzo dobrze sprawdzał się również w parze z adaptacyjnie dobieganym współczynnikiem uczenia 5.6.4.

5.6.4. Adaptacyjny współczynnik uczenia

Następnym krokiem jest dodanie mechanizmu, który pozwoli na adaptacyjny dobór współczynnika uczenia. Sieć przeucza się, gdyż w późniejszych epokach uczy się zbyt intensywnie istniejących już reprezentacji danych. Współczynnik uczenia kontroluje stopień nauki sieci i może być on ustawiany dynamicznie. Za pomocą wcześniej opisywanych *callback*’ów dodany został *LearningRateScheduler*, który będzie obniżał współczynnik uczenia wraz z postępem epok uczących. Listing 5.24 przedstawia regułę, według której będzie on modyfikowany.

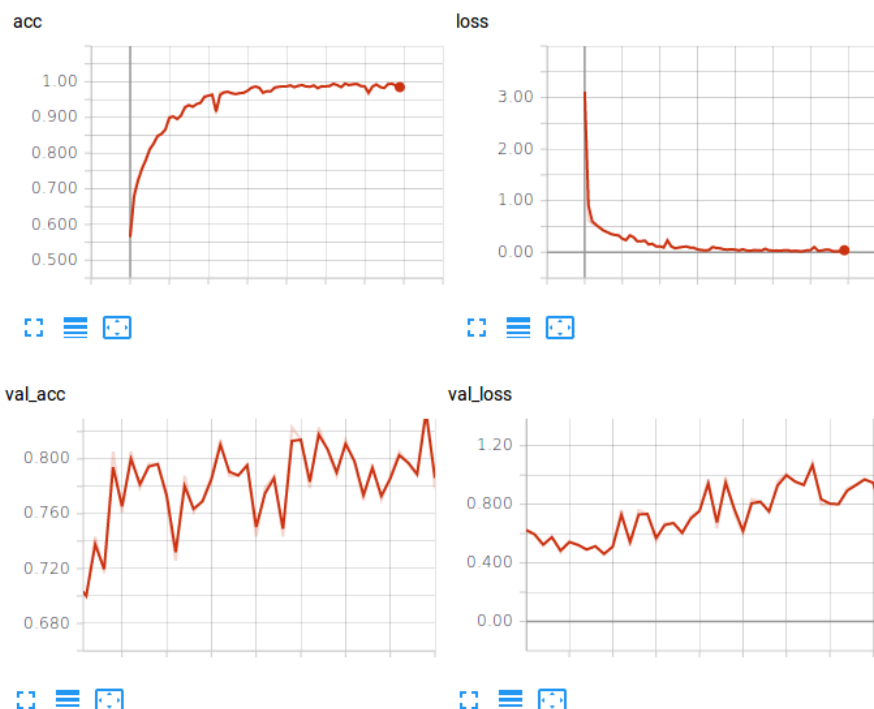
Dynamiczna zmiana współczynnika uczenia pozwoliła na częściowe zredukowanie nadmiernego dopasowania sieci, jednak wciąż zaobserwować można jej przeuczenie. Mimo to skuteczność została poprawiona do blisko **80,5%** 5.16.

Listing 5.24. Reguła modyfikacji współczynnika uczenia

```

1 def step_decay(epoch):
2     initial_lrate=0.1
3     drop=0.6
4     epochs_drop = 10.0
5     lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
6
7     return lrate

```



Rys. 5.16. Wyniki modelu po dodaniu zmiennego współczynnika uczenia

5.6.5. Modyfikacja rozmiaru sieci

Ostatnim wykonanym krokiem była modyfikacja rozmiaru sieci oraz poszczególnych warstw. Wprowadzanych było wiele zmian, na podstawie których można było zaobserwować, że zbyt mały rozmiar sieci zmniejszał skuteczność modelu, gdyż nie miał on wtedy wystarczającej przestrzeni do zapamiętania wszystkich ważnych informacji. Rozbudowywanie modelu początkowo zwiększało skuteczność, jednak po osiągnięciu pewnego progu dodawanie kolejnych neuronów oraz warstw przynosiło jedynie negatywne skutki.

Listing 5.25 zawiera model, który podczas testów osiągnął najlepsze rezultaty. Zawiera on 3 warstwy konwolucyjne z kolejno 64, 64 i 32 filtrami o rozmiarze 3x3. Po każdej z warstw zastosowano warstwę *BatchNormalization*, funkcję aktywacji *relu* oraz warstwę *max pooling*'u z filtrem o rozmiarze 2x2. W następstwie warstw konwolucyjnych sieć posiada 3 warstwy typu *fully connected* rozdzielonych warstwami *Dropout*. Warstwy posiadają rozmiar 64 oraz 32 neurony z warstwą wyjściową składającą się z jednego neuronu i sigmoidalną funkcją aktywacji (*ang. sigmoid*), która zwraca na wyjściu wartość 0 lub 1. Model skompilowany został przy użyciu

optymalizatora *SGD* i jako funkcja strat użyta została *binary_crossentropy*.

Listing 5.25. Ostateczny model sieci konwolucyjnej

```

1 model = Sequential()
2
3 model.add(Conv2D(64,(3,3),strides = (1,1),name='layer_conv1',padding='same',
4     input_shape=input_shape))
5 model.add(BatchNormalization())
6 model.add(Activation('relu'))
7 model.add(MaxPooling2D((2,2),name='maxPool1'))
8
9 model.add(Conv2D(64,(3,3),strides = (1,1),name='layer_conv2',padding='same'))
10 model.add(BatchNormalization())
11 model.add(Activation('relu'))
12 model.add(MaxPooling2D((2,2),name='maxPool2'))
13
14 model.add(Conv2D(32,(3,3),strides = (1,1),name='conv3',padding='same'))
15 model.add(BatchNormalization())
16 model.add(Activation('relu'))
17 model.add(MaxPooling2D((2,2),name='maxPool3'))
18
19 model.add(Flatten())
20 model.add(Dense(64,activation = 'relu',name='fc0'))
21 model.add(Dropout(0.25))
22 model.add(Dense(32,activation = 'relu',name='fc1'))
23 model.add(Dropout(0.25))
24 model.add(Dense(1, activation='sigmoid'))
25
26 model.compile(optimizer=SGD(lr=0.01, momentum=0.5, decay=0.0, nesterov=False),
27     loss='binary_crossentropy',
28     metrics=['acc'])

```

Podsumowanie modelu wygenerowane za pomocą funkcji *model.summary()* pokazujące użyte warstwy, rozmiary wyjściowe oraz ilość parametrów przedstawia się następująco 5.26.

Listing 5.26. Podsumowanie modelu

```

1 -----
2 Layer (type)                Output Shape                Param #
3 -----
4 layer_conv1 (Conv2D)         (None, 200, 16, 64)        640
5 -----
6 batch_normalization_88 (Batc (None, 200, 16, 64)        256
7 -----
8 activation_88 (Activation)   (None, 200, 16, 64)        0
9 -----
10 maxPool1 (MaxPooling2D)     (None, 100, 8, 64)         0
11 -----
12 layer_conv2 (Conv2D)         (None, 100, 8, 64)         36928
13 -----
14 batch_normalization_89 (Batc (None, 100, 8, 64)        256
15 -----
16 activation_89 (Activation)   (None, 100, 8, 64)         0
17 -----
18 maxPool2 (MaxPooling2D)     (None, 50, 4, 64)         0
19 -----
20 conv3 (Conv2D)               (None, 50, 4, 32)         18464
21 -----
22 batch_normalization_90 (Batc (None, 50, 4, 32)         128
23 -----
24 activation_90 (Activation)   (None, 50, 4, 32)         0
25 -----
26 maxPool3 (MaxPooling2D)     (None, 25, 2, 32)         0
27 -----
28 flatten_30 (Flatten)         (None, 1600)               0
29 -----
30 fc0 (Dense)                  (None, 64)                 102464
31 -----

```

```

32 dropout_59 (Dropout)          (None, 64)          0
33 -----
34 fc1 (Dense)                   (None, 32)          2080
35 -----
36 dropout_60 (Dropout)          (None, 32)          0
37 -----
38 dense_30 (Dense)              (None, 1)           33
39 =====
40 Total params: 161,249
41 Trainable params: 160,929
42 Non-trainable params: 320

```

Na listingu 5.27 zamieszczone zostały logi zawierające wyniki skuteczności sieci najlepszych modeli z poszczególnych *fold*’ów zebrane podczas trzech iteracji. Średnia skuteczność modelu to około **82%** (*grand mean of average accuracy*).

Listing 5.27. Wyniki skuteczności

```

1 Iteration 1
2
3 Fold: 0
4 --Best model validation accuracy: 79.65%
5
6 Fold: 1
7 --Best model validation accuracy: 83.93%
8
9 Fold: 2
10 --Best model validation accuracy: 83.04%
11
12 Fold: 3
13 --Best model validation accuracy: 84.82%
14
15 Fold: 4
16 --Best model validation accuracy: 86.61%
17
18 Fold: 5
19 --Best model validation accuracy: 77.68%
20
21 Fold: 6
22 --Best model validation accuracy: 81.08%
23
24 Fold: 7
25 --Best model validation accuracy: 78.38%
26
27 Fold: 8
28 --Best model validation accuracy: 81.98%
29
30 Fold: 9
31 --Best model validation accuracy: 83.78%
32
33
34 Best models average validation accuracy: 0.820943
35 Best models standard deviation of accuracy: 0.027431
36 Iteration 1 time: 511.17 seconds
37
38
39 Iteration 2
40
41 Fold: 0
42 --Best model validation accuracy: 78.76%
43
44 Fold: 1
45 --Best model validation accuracy: 83.04%
46
47 Fold: 2
48 --Best model validation accuracy: 83.93%
49
50 Fold: 3
51 --Best model validation accuracy: 84.82%
52
53 Fold: 4
54 --Best model validation accuracy: 87.50%

```

```
55
56 Fold: 5
57 --Best model validation accuracy: 75.89%
58
59 Fold: 6
60 --Best model validation accuracy: 77.48%
61
62 Fold: 7
63 --Best model validation accuracy: 83.78%
64
65 Fold: 8
66 --Best model validation accuracy: 78.38%
67
68 Fold: 9
69 --Best model validation accuracy: 84.68%
70
71
72 Best models average validation accuracy: 0.818264
73 Best models standard deviation of accuracy: 0.036667
74 Iteration 2 time: 527.52 seconds
75
76
77 Iteration 3
78
79 Fold: 0
80 --Best model validation accuracy: 81.42%
81
82 Fold: 1
83 --Best model validation accuracy: 85.71%
84
85 Fold: 2
86 --Best model validation accuracy: 83.93%
87
88 Fold: 3
89 --Best model validation accuracy: 84.82%
90
91 Fold: 4
92 --Best model validation accuracy: 86.61%
93
94 Fold: 5
95 --Best model validation accuracy: 76.79%
96
97 Fold: 6
98 --Best model validation accuracy: 77.48%
99
100 Fold: 7
101 --Best model validation accuracy: 81.98%
102
103 Fold: 8
104 --Best model validation accuracy: 82.88%
105
106 Fold: 9
107 --Best model validation accuracy: 84.68%
108
109
110 Best models average validation accuracy: 0.8263
111 Best models standard deviation of accuracy: 0.031417
112 Iteration 3 time: 643.31 seconds
113
114
115 ~~~Grand mean of average accuracy: 0.821836
116 ~~~Grand mean of standard deviation accuracy: 0.031839
```

5.7. Weryfikacja i ocena otrzymanych rezultatów

Przedstawione rezultaty były dotychczas sprawdzane jedynie na danych walidacyjnych. Prawdziwym sprawdzianem modelu jest zweryfikowanie jego skuteczności na wcześniej wydzielonym, odrębnym zbiorze testowym, który do tej pory nie brał

udziału w procesie nauki. Model, który nie jest przeuczony i posiada dobrą zdolność generalizacji powinien osiągnąć podobną skuteczność dla danych testowych.

W celu weryfikacji na danych testowych, do modelu sieci wczytane zostały wagi zapisane na dysku, które najlepiej radziły sobie podczas klasyfikacji na danych walidacyjnych. Proces sprawdzania skuteczności na danych testowych został wywołany osobno dla każdego modelu naczynego poszczególną kombinacją danych. Dzięki temu z uzyskanych wyników również można policzyć średnią, która będzie lepiej oddawała rzeczywisty stan, niż pojedynczy pomiar.

Na modelu wywołana została metoda *evaluate*, której parametrami były dane wejściowe oraz wyjściowe ze zbioru testowego. Wynikiem wykonania tych poleceń (listing 5.28) jest skuteczność modelu na danych, które zostały użyte po raz pierwszy (listing 5.29).

Listing 5.28. Sprawdzanie skuteczności modelu na danych testowych

```
1 model.load_weights("tmp/best_model.h5")
2 model.evaluate(x_test_3d, y_test, batch_size=16, verbose=0)
```

Listing 5.29. Skuteczność modelu na danych testowych

```
1 Iteration 1
2
3 Fold: 0
4 --Best model test accuracy: 77.97%
5
6 Fold: 1
7 --Best model test accuracy: 79.66%
8
9 Fold: 2
10 --Best model test accuracy: 79.66%
11
12 Fold: 3
13 --Best model test accuracy: 72.88%
14
15 Fold: 4
16 --Best model test accuracy: 79.66%
17
18 Fold: 5
19 --Best model test accuracy: 79.66%
20
21 Fold: 6
22 --Best model test accuracy: 74.58%
23
24 Fold: 7
25 --Best model test accuracy: 81.36%
26
27 Fold: 8
28 --Best model test accuracy: 76.27%
29
30 Fold: 9
31 --Best model test accuracy: 77.97%
32
33 Best models average test accuracy: 0.779661
34
35
36 Iteration 2
37
38 Fold: 0
39 --Best model test accuracy: 74.58%
40
41 Fold: 1
42 --Best model test accuracy: 79.66%
43
44 Fold: 2
45 --Best model test accuracy: 74.58%
46
```

```
47 Fold: 3
48 --Best model test accuracy: 74.58%
49
50 Fold: 4
51 --Best model test accuracy: 77.97%
52
53 Fold: 5
54 --Best model test accuracy: 72.88%
55
56 Fold: 6
57 --Best model test accuracy: 67.80%
58
59 Fold: 7
60 --Best model test accuracy: 71.19%
61
62 Fold: 8
63 --Best model test accuracy: 79.66%
64
65 Fold: 9
66 --Best model test accuracy: 74.58%
67
68 Best models average test accuracy: 0.747458
69
70
71 Iteration 3
72
73 Fold: 0
74 --Best model test accuracy: 79.66%
75
76 Fold: 1
77 --Best model test accuracy: 74.58%
78
79 Fold: 2
80 --Best model test accuracy: 76.27%
81
82 Fold: 3
83 --Best model test accuracy: 77.97%
84
85 Fold: 4
86 --Best model test accuracy: 79.66%
87
88 Fold: 5
89 --Best model test accuracy: 76.27%
90
91 Fold: 6
92 --Best model test accuracy: 72.88%
93
94 Fold: 7
95 --Best model test accuracy: 83.05%
96
97 Fold: 8
98 --Best model test accuracy: 77.97%
99
100 Fold: 9
101 --Best model test accuracy: 81.36%
102
103 Best models average test accuracy: 0.779661
104
105
106 ~~~Grand mean of average test accuracy: 0.768927
```

Osiągnięte wyniki nie są identyczne jak te uzyskane na danych walidacyjnych, jednak są one wystarczająco podobne, aby stwierdzić, że model dobrze radzi sobie z nowymi danymi. Wynik na poziomie prawie **77%** (76.89%) na danych testowych jest zadowalający i przekracza założony próg 70% ustalony w rozdziale 3.3.

Rozdział 6

Podsumowanie i wnioski końcowe

Struktura sieci utworzona podczas przeprowadzonych badań osiągnęła skuteczność na poziomie przewyższającym początkowe założenia. Można więc stwierdzić, że cel pracy został osiągnięty. Pomyślna próba wykorzystania głębokich sieci neuronowych w tym konkretnym problemie jest kolejnym potwierdzeniem tego, że dziedzina *deep learning*'u niesie za sobą bardzo duże możliwości.

Ze względu na dużą popularność uczenia głębokiego, z dnia na dzień powstają nowe architektury sieci oraz metody ich nauki. W czasie przeprowadzonych badań nie zostały przetestowane wszystkie dostępne opcje, dlatego wciąż istnieje możliwość ulepszenia otrzymanych rezultatów. Kilka metod, które mogą zostać wzięte pod uwagę, opisano w rozdziale 6.1.

Wypracowany model osiągnął zadowalający poziom skuteczności na danych testowych, jednak dodatkowym sprawdzianem jego efektywności byłaby ponowna weryfikacja na zupełnie innym zbiorze danych np. nowych pacjentów ze Szpitala Uniwersyteckiego oraz porównanie wskazań sieci neuronowej z wiedzą wykwalifikowanego neurologa. Alternatywnym zbiorem danych, który może zostać wzięty pod uwagę podczas dodatkowej weryfikacji jest wspomniany w rozdziale 3.3 publicznie dostępny zbiór CHB-MIT (Children's Hospital of Boston-Massachusetts Institute of Technology).

6.1. Możliwości rozwoju

Choć osiągnięty został cel pracy, w przyszłości może zostać podjęta próba udoskonalenia wyników. Działania, które mogą być wykonane to m.in.:

- Próba zwalczania zjawiska nadmiernego dopasowania dla zaproponowanego modelu,
- Przetworzenie danych do innej postaci (*ang. feature extraction*) np. wizualizacja aktywności mózgu opisana w pozycji [10] lub stworzenie tzw. *heat map*'y metodą *matching pursuit* przedstawiona w książce [14],
- Wykorzystanie gotowych modeli sieci wbudowanych w bibliotekę *Keras* np. *Xception*, *ResNet*, *VGG19* itd. opisane w dokumentacji Keras [15].

Dodatek A

Użycie biblioteki Keras w języku R

Biblioteka Keras najczęściej używana jest w połączeniu z językiem Python, jednak istnieje również możliwość wykorzystania jej w języku R. Pomimo, że podczas pisania pracy wykorzystany został Python z uwagi na jego popularność, to poczyniono również pewne kroki w celu sprawdzenia w jaki sposób użyć biblioteki Keras w języku R.

W niniejszym dodatku przedstawione zostanie jak stworzyć prosty model sieci neuronowej, która zostanie nauczona na przykładowych danych. Dodatkowo zaprezentowane zostanie jak wygląda definicja zaproponowanego modelu dla problemu rozpoznawania napadów padaczkowych na podstawie odczytów z EEG (rozdział 5.6) w języku R i środowisku RStudio.

A.1. Implementacja przykładowego modelu do klasyfikacji danych MNIST

Po uruchomieniu środowiska RStudio należy zainstalować bibliotekę keras (patrz listing A.1).

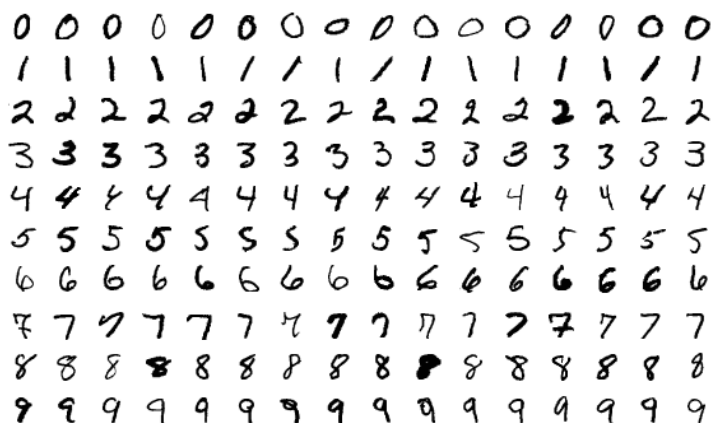
Listing A.1. Instalacja Keras

```
1 devtools::install_github("rstudio/keras")
2
3 library(keras)
4 install_keras()
```

W tym przykładzie zostanie użyty klasyczny zbiór danych MNIST, który zawiera obrazki w skali szarości o rozmiarach 28x28 pixeli przedstawiające odręcznie pisane cyfry wraz z odpowiadającymi im etykietami (patrz rysunek A.1). Zbiór MNIST dostępny jest w bibliotece Keras.

W celu wczytania wartości wystarczy użyć funkcji `dataset_mnist()` i przypisać wartości do zmiennej, a następnie wydzielić odpowiednie zbiory treningowe oraz testowe (patrz listing A.2).

Stworzony zostanie prosty model sieci neuronowej typu *fully connected*, która wymaga przekształcenia danych do postaci wektorów. Przygotowane dane wejściowe są w postaci 3-wymiarowej tablicy, więc należy zredukować liczbę wymiarów oraz



Rys. A.1. Przykładowe dane ze zbioru MNIST

Listing A.2. Wczytywanie danych MNIST

```

1 mnist <- dataset_mnist()
2 x_train <- mnist$train$x
3 y_train <- mnist$train$y
4 x_test <- mnist$test$x
5 y_test <- mnist$test$y

```

dodatkowo przeskalować, aby znalazły się one w przedziale $<0;1>$ (patrz listing A.3).

Listing A.3. Przygotowanie danych wejściowych

```

1 x_train <- array_reshape(x_train, c(nrow(x_train), 784))
2 x_test <- array_reshape(x_test, c(nrow(x_test), 784))
3
4 x_train <- x_train / 255
5 x_test <- x_test / 255

```

Dane wyjściowe należy zakodować za pomocą kodu "1 z n" (*ang. one-hot encoding*) (patrz listing A.4).

Listing A.4. Przygotowanie danych wyjściowych

```

1 y_train <- to_categorical(y_train, 10)
2 y_test <- to_categorical(y_test, 10)

```

Utworzony został model składający się z dwóch warstw ukrytych oraz jednej wyjściowej, zawierający odpowiednio 256, 128 i 10 neuronów. Na ostatniej warstwie użyta została funkcja aktywacji *softmax*, która zwróci prawdopodobieństwo zajęcia jednego z 10 stanów (patrz listing A.5).

Po utworzeniu modelu wywoływana jest metoda *fit()*, która rozpoczyna proces uczenia (patrz listing A.6). Zastosowany został podział na zbiory uczący i walidacyjny w stosunku 4:1.

Sprawdzenie skuteczności modelu na danych testowych odbywa się za pomocą funkcji *evaluate()* pokazanej na listingu A.7.

Listing A.5. Utworzenie modelu

```

1 model <- keras_model_sequential()
2 model %>%
3   layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
4   layer_dropout(rate = 0.4) %>%
5   layer_dense(units = 128, activation = 'relu') %>%
6   layer_dropout(rate = 0.3) %>%
7   layer_dense(units = 10, activation = 'softmax')
8
9 model %>% compile(
10   loss = 'categorical_crossentropy',
11   optimizer = optimizer_rmsprop(),
12   metrics = c('accuracy')
13 )

```

Listing A.6. Rozpoczęcie procesu uczenia

```

1 history <- model %>% fit(
2   x_train, y_train,
3   epochs = 50, batch_size = 64,
4   validation_split = 0.2
5 )

```

Listing A.7. Sprawdzenie modelu na danych testowych

```

1 model %>% evaluate(x_test, y_test, verbose = 0)
2
3
4 $loss
5 [1] 0.1241
6
7 $acc
8 [1] 0.9788

```

A.2. Model sieci do rozpoznawania stanów padaczkowych w języku R

Na listingu A.8 zaprezentowany został model konwolucyjnej sieci neuronowej utworzony w języku R odzwierciedlający sieć użytą do rozpoznawania stanów padaczkowych przedstawioną w rozdziale 5.6.

Listing A.8. Model konwolucyjnej sieci neuronowej do rozpoznawania stanów padaczkowych

```

1 model <- keras_model_sequential() %>%
2
3   layer_conv_2d(filters = 64, kernel_size = c(3, 3), input_shape = input_shape) %>%
4   layer_batch_normalization() %>%
5   layer_activation("relu") %>%
6   layer_max_pooling_2d(pool_size = c(2, 2)) %>%
7
8   layer_conv_2d(filters = 64, kernel_size = c(3, 3), input_shape = input_shape) %>%
9   layer_batch_normalization() %>%
10  layer_activation("relu") %>%
11  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
12
13  layer_conv_2d(filters = 32, kernel_size = c(3, 3), input_shape = input_shape) %>%
14  layer_batch_normalization() %>%
15  layer_activation("relu") %>%
16  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
17

```

```
18     layer_flatten() %>%
19     layer_dense(units = 64, activation = "relu") %>%
20     layer_dropout(rate=0.25) %>%
21     layer_dense(units = 32, activation = "relu") %>%
22     layer_dropout(rate=0.25) %>%
23
24     layer_dense(units = 1, activation = "sigmoid")%>% compile(
25         optimizer=optimizer_sgd(lr=0.01, momentum=0.5, decay=0.0, nesterov=False),
26         loss='categorical_crossentropy',
27         metrics='accuracy')
```

A.3. Podsumowanie

Z pewnością biblioteka Keras może być z powodzeniem używana również w języku R, gdyż oferuje ona takie same możliwości, jak dla języka Python. Należy jednak pamiętać, że używanie Keras'a w połączeniu z językiem Python jest o wiele bardziej popularne. Z tego powodu istnieje o wiele więcej poradników oraz artykułów opisujących wykorzystanie właśnie tej kombinacji narzędzi, co może być pomocne przy implementacji bardziej złożonych rozwiązań.

Dodatek B

Płyta DVD

Do tekstu pracy załączona została płyta DVD z następującą zawartością:

- plik **/praca-dyplomowa.pdf** - tekst pracy dyplomowej,
- katalog **/projekt/** - zawiera wszystkie pliki wykonanego projektu,
- katalog **/oprogramowanie/** - zawiera oprogramowanie wymagane do uruchomienia projektu. W szczególności są to:
 - katalog **/oprogramowanie/docker/** - program Docker do tworzenia wirtualnych kontenerów, umożliwiający uruchomienie całego środowiska programistycznego

Bibliografia

- [1] engmrk.com. *Module 22 – Implementation of CNN Using Keras*.
<https://engmrk.com/module-22-implementation-of-cnn-using-keras/>.
- [2] wikipedia.com. *10-20 system (EEG)*.
[https://en.wikipedia.org/wiki/10%E2%80%9320_system_\(EEG\)](https://en.wikipedia.org/wiki/10%E2%80%9320_system_(EEG)).
- [3] Francois Chollet. *Deep Learning with Python*. 2018.
- [4] Jacob M. Williams. *Deep Learning and Transfer Learning in the Classification of EEG Signals*. 2017.
- [5] Marvin Minsky and Seymour Paper. *Perceptrons*. 1969.
- [6] Yoshua Bengio Ian Goodfellow and Aaron Courville. *Deep learning*. Book in preparation for MIT Press, 2016.
- [7] Andrew Lim Pierre Thodoroff, Joelle Pineau. *Learning Robust Features using Deep Learning for Automatic Seizure Detection*. 2016.
- [8] Emad-ul-Haq Qazi Ihsan Ullah, Muhammad Hussain and Hatim Aboalsamh. *An Automated System for Epilepsy Detection using EEG Brain Signals based on Deep Learning Approach*.
- [9] Nick Hershey. *Detecting Epileptic Seizures in Electroencephalogram Data*.
- [10] Andrew Lim Pierre Thodorof, Joelle Pineau. *Learning Robust Features using Deep Learning for Automatic Seizure Detection*. arXiv:1608.00220v1 [cs.LG] 31 Jul 2016.
- [11] Epilepsy foundation. *About Epilepsy: The Basics*.
<https://www.epilepsy.com/learn/about-epilepsy-basics>.
- [12] Tensorflow. *Tensorboard docs*.
<https://github.com/tensorflow/tensorboard>.
- [13] Christian Szegedy Sergey Ioffe. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv:1502.03167v3 [cs.LG] 2 Mar 2015.
- [14] Rafał Scherer-Ryszard Tadeusiewicz Lotfi A. Zadeh Jacek M. Zurada (Eds.) Leszek Rutkowski, Marcin Korytkowski. *Artificial Intelligence and Soft Computing*. 2013.
- [15] Keras. *Keras docs*. <https://keras.io/applications/>.