Uniwersytet Zielonogórski

Wydział Informatyki, Elektrotechniki i Automatyki

Praca magisterska

Kierunek: Informatyka

Wykorzystanie głębokich sieci neuronowych do wspomagania diagnostyki medycznej

Krystian Dziędzioła

Promotor:
dr 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 z powodzeniem zastosować do 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

Spis rysunków

Spis tabel

Rozdział 1

Wstęp

1.1. Wprowadzenie

Współczesna technologia sztucznej inteligencji, 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.

W tej pracy zostanie sprawdzona możliwość zastosowania głębokich sieci neuronowych do diagnostyki napadów padaczkowych na podstawie odczytów z elektoencefalogramu (EEG).

1.2. Cel i zakres pracy

Celem pracy jest próba wykorzystania głębokich sieci neuronowych do wspomagania diagnostki 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 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 ze sztuczną inteligencją, uczeniem maszynowym oraz klasycznymi i głębokimi sieciami neuronowymi została oparta na kilku pozycjach obszernie opisujących tę tematykę. W szczególności są to: [2], [3], [4] oraz [7].

Druga część opisująca praktyczne zastosowanie głębokich sieci neuronowych została oparta głównie na wiedzy zdobytej z książki [2] 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. [10], [14], [11] oraz [9].

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 zagadnienie sztucznej inteligencji, uczenia maszynowego oraz klasycznych i głębokich sieci neuronowych. Rozdział 3 przedstawia opis wybranego problemu z zakresu diagnostyki medycznej, dla którego zostanie podjęta próba rozwiązania przy pomocy głęgokich sieci neuronowych.

Druga część składa się z rozdziałów 4 oraz 5, które zawierają przegląd dostępnych bibliotek oraz narzędzi programistycznych, opis implementacji procesu uczenia oraz przeprowadzonych badań w celu sprawdzenia możliwości rozwiązania wybranego problemu.

Rozdział 2

Wprowadzenie do głębokich sieci neuronowych

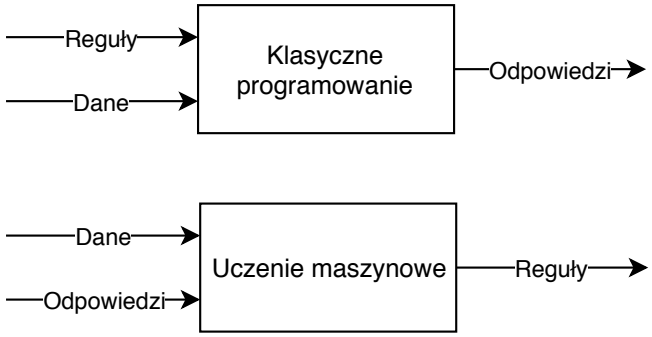
2.1. Sieci neuronowe a sztuczna inteligencja

W ciągu ostatnich kilku lat sztuczna inteligencja stała się tematem dużego zainteresowania w nauce, technologii oraz życiu codziennym. Ma to miejsce za sprawą szerokiego spektrum jej zastosowań oraz możliwości wykorzystania do zadań, które do tej pory były niemal nieosiągalne przez maszyny. Są to m.in. autonomiczne pojazdy, inteligentni wirtualni asystenci, wspomaganie diagnostyki medycznej oraz wiele innych.

Sztuczna inteligencja datuje swoje początki na lata 50-te XX wieku kiedy to zaczęto prowadzić badania nad tym czy komputer jest w stanie nauczyć się "myśleć". Bardziej ścisłą definicją celu prowadzonych badań mogłaby być "próba automatyzacji zadań intelektualnych, które normalnie wykonywane są przez człowieka". Początkowo próby te były poczynione poprzez zaprogramowanie szeregu reguł, którymi miała posługiwać się maszyna w podejmowaniu swoich własnych decyzji. Jest to tzw. symboliczna sztuczna inteligencja *(ang. symbolic artificial intelligence)*, czyli klasyczne podejście polegające na zaprogramowaniu określonego algorytmu działania. [5] Zauważono jednak, że zbiór określonych przez programistów reguł sprawdza się do rozwiązywania dobrze zdefiniowanych problemów logicznych takich jak np. gra w szachy. Takie podejście nie jest jednak wystarczające do osiągnięcia przez komputer umiejętności podejmowania decyzji, która mogłaby symulować ludzką inteligencję oraz wykonwania przez nią bardziej abstrakcyjnych zadań takich jak np. rozpoznawanie mowy, obrazu lub tłumacznie języków.

Nasuwa się pytanie czy komputer jest w stanie wyjść poza ramy ściśle zdefiniowanych reguł i samemu nauczyć się w jaki sposób wykonywać określone zadania? Odpowiedzią na to pytanie oraz niedoskonałości klasycznego programowania było zastosowanie podejścia zwanego dziś uczeniem maszynowym *(ang. machine learning)*. Polega ono na tym, że maszynie nie jest dostarczany ściśle określony algorytm działania, a jedynie przedstawiane są dane oraz wyniki jakie powinny powstać po przetworzeniu tychże danych. Maszyna ma za zadanie stworzyć swoje własne reguły, które będą odzwierciedlać sposób w jaki prawidłowo należy wykonać konkretne zadanie. Można więc powiedzieć, że w uczeniu maszynowym, maszyna jest uczona, a nie programowana. Pomimo tego, że uczenie maszynowe zaczęło się roz-

wijać dopiero w latach 90-tych bardzo szybko stało się popularne za sprawą tego, że przynosiło najlepsze wyniki spośród wszystkich podejść w dziedzinie sztucznej inteligencji. Różnicę pomiędzy klasycznym programowaniem a uczeniem maszynowym zaprezentowano na rysunku 2.1.



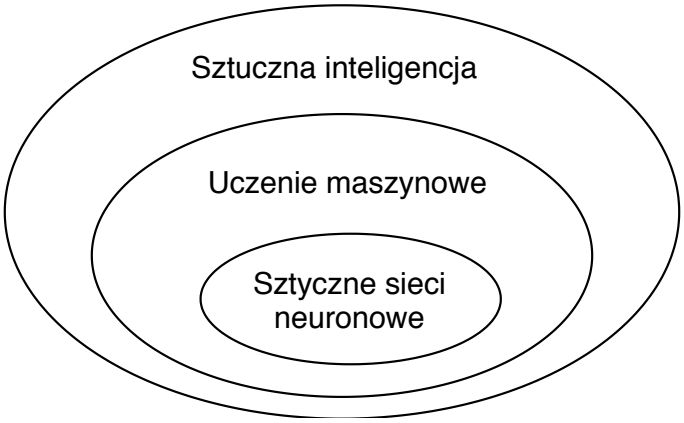
Rys. 2.1. Porównanie klasycznego programowania z metodą uczenia maszynowego

Mnogości zastosowań uczenia maszynowego dorównuje również ilość algorytmów, które są wykorzystywane w tym podejściu. Dostępnych jest wiele modeli, poczynając od bardzo prostych, kończąc na dosyć skomplikowanych. Kilka najczęściej używanych, to m.in.:

- **drzewa decyzyjne** *(ang. decision trees)*,
- **algorytm k-średnich** *(ang. k-means)*,
- **sztuczne sieci neuronowe** *(ang. artificial neural networks)*. [6]

Przedmiotem tej pracy jest ostatnia z wymienionych metod, czyli sztuczne sieci neuronowe. Jest to zestaw algorytmów, który został zaprojektowany w sposób inspirowany działaniem ludzkiego mózgu. W rzeczywistości jednak, ich działanie nie jest do końca zgodne z tym, w jaki sposób działa umysł człowieka. Sieci neuronowe potrafią przyjmować dane w formie wektorów liczbowych, które mogą zawierać takie informacje jak np. obrazy, dźwięki, tekst lub szeregi czasowe. Na ich podstawie są w stanie dokonywać klasyfikacji, klasteryzacji oraz predykcji. Jednym z typów sztucznych sieci neuronowych są tzw. głębokie sieci neuronowe *(ang. deep neural networks)*, które znacząco zwiększają możliwości klasycznych sieci neuronowych. Zostaną one omówione w rozdziale 2.3.

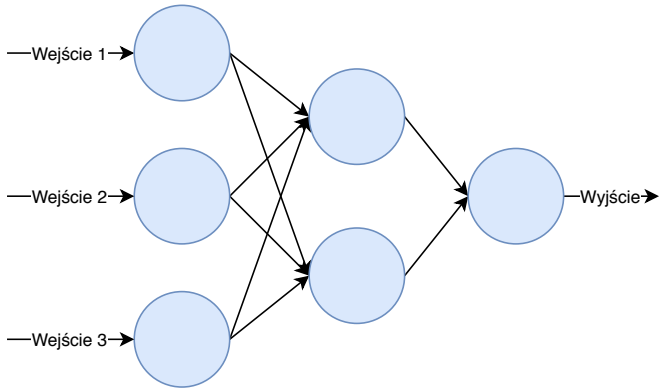
Można zauważyć, że sztuczna inteligencja jest pojęciem bardzo szerokim. W jej skład wchodzą różne metody m.in. uczenie maszynowe. Z kolei specyficznym podejściem w uczeniu maszynowym są sieci neuronowe. Zależności pomiędzy tymi pojęciami przedstawia rysunek 2.2.



Rys. 2.2. Zależność między sztuczną inteligencją, uczeniem maszynowym i sieciami neuronowymi

2.2. Klasyczne sieci neuronowe

Sieci neuronowe to dziedzina uczenia maszynowego, która jako model matematyczny podlegający uczeniu wykorzystuje sturukturę sieci składającą się z wielu jednostek obliczeniowych zwanych neuronami. Neurony wykonują podstawowe obliczenia i ich wyniki przekazują do kolejnych neuronów. Operacje, które są przez nie wykonywane to przeważnie sumowanie otrzymanych informacji oraz zastowanie prostej, nieliniowej funkcji. W większości sieci neuronowych, neurony zgrupowane są w tzw. warstwy. Obliczenia wykonane przez jedną warstwę są przekazywane do kolejnej, która z kolei wykonuje obliczenia na otrzymanych wynikach. Ostatnia warstwa zwraca końcowy wynik, który jest interpretowany w różny sposób w zależności od wykonywanej operacji np. klasyfikacja lub regresja. Schemat prostej sieci neuronowej przedstawiony został na rysunku 2.3



Rys. 2.3. Struktura prostej sieci neuronowej

Pierwszą koncepcję neuronu datuje się na rok 1943, w którym to powstał model neuronu McCulloch-Pitts. Był on bardzo prosty w porównaniu do współczesnych sieci neuronowych, gdyż pozwalał jedynie na wykorzystanie wartości binarnych na wyjściu z neuronów. Każdy z nich sumował wartości wejściowe i przyrównywał je do zera. Dodatkowo, nie istniała żadna reguła aktualizacji wewnętrznych wartości neuronów (tzw. wag), która jest niezbędna do prawidłowego przebiegu procesu adaptacji neuronu do nowych informacji. Bez takiej reguły wszystkie wagi neuronów

musiałyby być ustawiane ręcznie.

W latach 50-tych przedstawiony został perceptron, który jest najprostszą siecią neuronową składającą się wielu neuronów McCullocha-Pittsa. Implementuje on algrotym uczenia nadzorowanego klasyfikatorów binarnych. Jest funkcją przynależności potrafiąca przydzielić jedną z dwóch klas do danych parametrów wejściowych. W perceptronie została przedstawiona prosta reguła służąca do aktualizacji wag dla kolejnych iteracji. Wzór na wagę w kroku $t + 1$ przedstawia się następująco:

$$w_i(t+1) = w_i(t) + (d_j - y_j(t))x_{j,i} \tag{2.1}$$

gdzie w_i to waga i-tego neuronu, d_j to oczekiwana wartość dla j-tego wejścia, $y_i(t)$ oznacza wartość obliczoną z j-tego wejścia, natomiast $x_{j,i}$ jest i-tą wartością j-tego wejścia. Oznacza to, że przyszła wartość wagi neuronu obliczona jest przez dodanie błędu (różnicy między wartością oczekiwaną a obliczoną) pomnożonego przez wartość rzeczywistą do wartości aktualnej wagi. Reguła ta może być stosowana jedynie do uczenia jednowarstwowych sieci neuronowych, co znacznie ogranicza zakres jej zastosowań. Perceptron może więc być używany jedynie do problemów liniowo separowalnych.[3]

W latach 60-tych zostało matematycznie udowodnione, że sieć z pojedynczą warstwą nie posiada możliwości klasyfikowania problemów, które nie są liniowo separowalne. Przykładem jest funkcja alternatywy wykluczającej - XOR. Kolejne, bardziej skomplikowane problemy, których się podejmowano, takie jak m.in. rozpoznawanie mowy, kończyły się więc niepowodzeniem. Możliwość uczenia sieci wielowarstwowych była więc konieczna w dalszym rozwoju sieci neuronowych. W tej samej publikacji [4] zostało również udowodnione, że 2-warstwowa sieć jest w stanie zamodelować niemal każdą funkcją. W praktyce jest to jednak ciężkie do osiągnięcia.

W późniejszym okresie przedstawiony został algorytm wstecznej propagacji błędów *(ang. backpropagation)*, który umożliwiał uczenie wielowarstwowych sieci neuronowych. Oparty jest on na minimalizacji wartości tzw. funkcji straty z wykorzystaniem optymalizacyjnej metody największego spadku. Błędy obliczane są na warstwie wyjściowej i przekazywane są wstecz, do warstw poprzedzających. Jest to możliwe dzięki wykorzystaniu tzw. metody łańcuchowej *(ang. chain rule)*. Szczegóły dotyczące tego algorytmu zostały opisane w wielu publikacjach m.in. w [7]. Dzięki zastosowaniu tego typu podejścia osiągnięty został cel, który zakładał umożliwienie uczenia wielowarstwowych sieci neuronowych. Pozwoliło to w znaczącym stopniu przyspieszyć rozwój w tej dziedzinie.

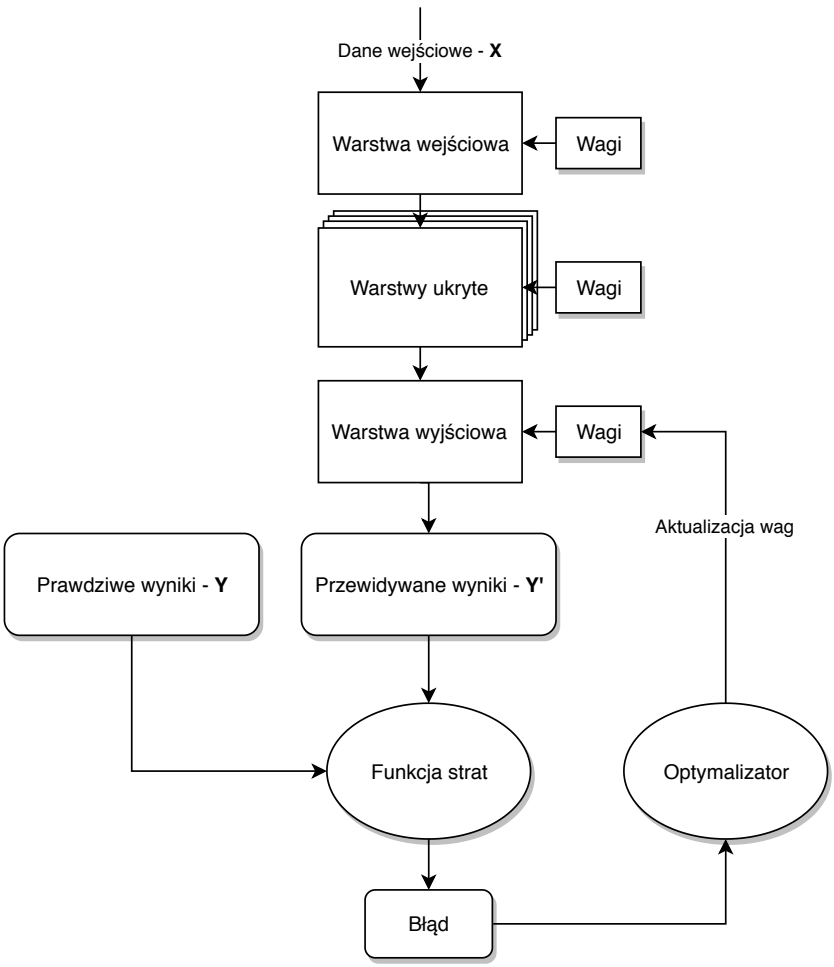
Algorytm wstecznej propagacji błędów stał się najpopularniejszym i najbardziej skutecznym algorytmem do nauki sieci neuronowych. Ma on właśnie zastosowanie w głębokich sieciach neuronowych, które zostały opisane w rozdziale 2.3.

2.3. 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 naukę kolejnych warstw reprezentacji, które z każdą warstwą są coraz bardziej konkretne. 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 uczą się dziesiątek, a czasem nawet setek kolejnych warst reprezentacji danych. Każda z nich uczy się samodzielnie na podstawie przedstawionych im informacji.

Inne podejścia, takie jak np. klasyczne uczenie maszynowe skupia się na nauce jedynie jednej lub dwóch warstw reprezentacji danych, dlatego tego typu metody są często nazywane "płytkim uczeniem" *(ang. shallow learning)*. W deep learning'u jako model matematyczny prawie zawsze używa się sieci neuronowych, które składają się z nałożonych na siebie wielu warstw neuronów.



Rys. 2.4. Schemat procesu uczenia głębokiej sieci neuronowej

Każda z warstw posiada zestaw tzw. wag *(ang. weights)*, które określają sposób w jaki interpretowane są dane wejściowe. Z technicznego punktu widzenia wagi są parametrami transformacji, które są wykonywane przez warstwy na danych wejściowych. W tym kontekście uczenie oznacza poszukiwanie odpowiedniego zestawu wag, który pozwoli na poprawne przekształcanie danych wejściowych w oczekiwane rezultaty. W głębokich sieciach neuronowych mogą istnieć miliony tego typu parametrów, więc znalezienie odpowiedniej wartości dla każdego z nich może być trudnym zadaniem, biorąc pod uwagę, że zmiana jednego parametru może mieć wpływ na zachowanie innych.

W celu poprawnej modyfikacji parametrów sieć musi obserwować wyniki wyjściowe i porównywać je z wynikami docelowymi, które dostarczone zostały w procesie

uczenia. Pomiarem tego jak bardzo uzyskane wyniki różnią się od tych prawdziwych zajmuje się tzw. funkcja strat *(ang. loss function)*. Zwraca ona konkretną wartość liczbową popełnionego błędu, który następnie może być minimalizowany w celu uzyskiwania coraz to lepszych wyników. Odbywa się to poczynając od warstwy ostatniej, kończąc na warstwie wejściowej stosując algorytm wstecznej propagacji błędów, który implementowany jest przez tzw. optymalizator *(ang. optimizer)*. [7]

Początkowo wagi posiadają losowe wartości, które dostosowywane są w ramach wielu iteracji zwanych epokami uczącymi *(ang. epochs)*. Przeważnie proces powtarzany jest dziesiątki razy na tysiącach przykładowych danych. Wybierany jest zestaw wag sieci, który posiada najmniejszą wartość funkcji strat, czyli jest najlepiej dostosowany do poprawnego przetwarzania danych wejściowych na wyjściowe. Cały proces uczenia głęgokiej sieci neuronowej został zaprezentowany na rysunku 2.4.

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)*.

2.4. Architektury głębokich sieci neuronowych

Wyróżnia się 5 najczęściej stosowanych architektur głębokich sieci neuronowych. Są to:

- **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)*

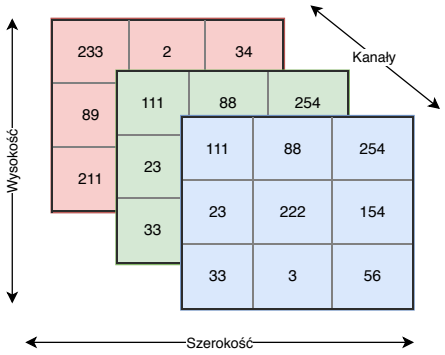
Zostaną omówione jedynie dwie pierwsze architektury, gdyż to one znalazły zastosowanie w niniejszej pracy.

2.4.1. 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.

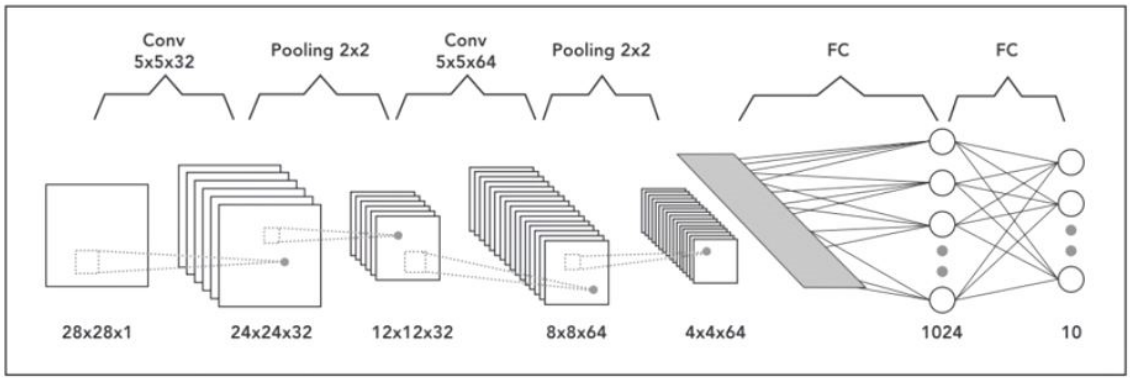
Przewagą tego typu sieci nad klasycznymi sieciami neuronowymi jest możliwość rozpoznawania tymczasowych oraz przestrzennych zależności pomiędzy danymi. Jest to możliwe, dzięki temu, że dane które trafiają do sieci mogą posiadać strukturę wielowymiarową. Przykładowo, obrazki zapisane w formacie RGB są 3-wymiarową macierzą przechowującą informację o wartościach koloru każdego z pikseli 2.5.

Na danych przestrzennych wykonywane są operacje takie jak: konwolucja *(ang. convolution)*, zmniejszanie rozmiaru danych *(ang. subsampling)* oraz aktywacja *(ang.*



Rys. 2.5. Dane liczbowe obrazka w formacie RGB

activation). Znalezione cechy są następnie podawane na wejście klasycznej sieci, gdzie neurony połączone są każdy z każdym *(ang. fully connected)*, która wykonuje ostateczne operacje 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. w [2] oraz [7]. Przykładowy schemat splotowej sieci neuronowej składający się z kliku warstw konwolucyjnych połączonych z warstwami zmniejszającymi rozmiar danych (tzw. *pooling*'iem), która została zakończona dwoma warstwami sieci *FC - fully connected* przedstawiony jest na rysunku 2.6.



Rys. 2.6. Schemat działania sieci splotowej [1]

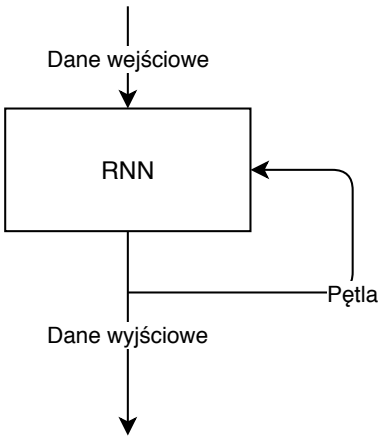
Zaletą tego typu sieci jest bez wątpienia możliwość przetwarzania wielowymiarowych danych, dzięki czemu są one w stanie odnajdywać przestrzenne zależności pomiędzy nimi, co przekłada się na ich wysoką skuteczność i wszechstronność.

2.4.2. Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe są często używane do rozwiązywania problemów, gdzie wykorzystywane są dane sekwencyjne. W praktyce znajdują one zastosowanie w takich problemach jak np. synteza mowy, prognozowanie pogody czy automatyczne tłumaczenie języków.

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 pętli zaprezentowany został na rysunku 2.7.



Rys. 2.7. Pętla w rekurencyjnej sieci neuronowej

Zaletą rekurencyjnych sieci neuronowych jest to, że współdzielą one jeden zestaw parametrów we wszystkich krokach, dzięki temu pozwala to zredukować ich ilość. Mogą być on również używane w połączeniu z sieciami splotowymi.

Do wad należy m.in. to, że nie są one w stanie zapamiętywać zbyt długich sekwencji. Dodatkowo, ich głębokość nie może być zbyt duża, ponieważ używane funkcje aktywacji powodują, że gradient zmian zanika po przejściu przez kilka warstw.

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 [8].

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 charakterystki 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 zróznicowanie danych. Sposób, oraz wyniki przeprowadzonych w ten sposób badań można znaleźć m.in. w [9], [10] oraz [11].

3.2. Dostępne dane

Badania opisane w niniejszej pracy zostały przeprowadzone na danych udostępnionych przez Szpital Wojewódzki 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 dostępnea 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 rozwiąznie można z powodzeniem zastosować również do posiadanej bazy odczytów. Wyniki przewyższające **70%** skuteczności powinny być dobrym wyznacznikiem pomyślności wykonanych badań.

Rozdział 4

Przegląd dostępnych bibliotek oraz narzędzi programistycznych

4.1. Dostępne narzędzia

Istnieje niezliczona ilość 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 biblitek (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 umożliwa 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** - bilbioteka umożliwiająca wykonywanie obliczeń na karcie graficznej podczas nauki sieci neuronowych. Jej używanie jest bardzo zalecane, gdyż umożliwia o wiele szybsze przetwarzanie danych. Wymagane jest posiadania 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

Próba rozwiązania problemu

5.1. Proces uczenia

Proces uczenia sieci neuronowej stworzony został w środowisku Jupyter Notebook, które umożliwia 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 5.1.1
- podział danych na zbiory służące do nauki oraz testowania przy pomocy k-krotnej walidacji krzyżowej *(ang. k-fold cross-validation)* 5.1.2
- utworzenie modelu sieci neuronowej 5.1.3.1
- rozpoczęcie uczenia modelu 5.1.3
- sprawdzenie skuteczności i przedstawienie wyników 5.1.3.4

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

5.1.1. 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 odpowiadające konkretnym kanałom (*EEG_FP1_F3*, *EEG_FP2_F4* itd.). Ostatnia kolumna (o nagłówku t) przedstawia czas w sekundach, w którym miał miejsce pomiar.

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 z danymi wygląda następująco:

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_O1"     "EEG_P4_O2"     "EEG_FP1_F7"    "EEG_FP2_F8"
      "EEG_F7_T3"     "EEG_F8_T4"     "EEG_T3_T5"     "EEG_T4_T6"     "EEG_T5_O1"
      "EEG_T6_O2"     "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
```

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

```
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
```

W celu wykorzystania przedstawionych danych w procesie uczenia sieci neuronowej 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.pckl'
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,
                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
            input_variable_file:
93          pickle.dump([input_data, target, headers], input_variable_file)
94
95      del input_data, target, headers
96
97
98  # Normalizuje dane używając obiektu StandardScaler(). Zwrócone dane posiadają średnią 0
        oraz odchylenie standardowe 1.
99  def normalize(x, y):
100     scalers = {}
101     for channel_number in range(x.shape[1]):
102         scalers[channel_number] = StandardScaler()
103         x[:, channel_number, :] = scalers[channel_number].fit_transform(x[:,
                channel_number, :])
104     return x, y.astype(int)
105
106
107 # Pobiera zmienne z danymi z utworzonego pliku tymczasowego
108 def load_input_data(chunk_size_in_seconds):
109     with open(INPUT_DATA_FILE_PATH.format(chunk_size_in_seconds), 'rb') as
            input_data_file:
110         input_data, target, headers = pickle.load(input_data_file)
111
```

```
112     return input_data, target, headers
113
114
115 # Pobiera dane wykorzystując funkcję load_input_data() i wywołuje funkcję
        prepare_chunks() z pliku chunks_creator.py w celu utworzenia okien czasowych z
        danymi. Utworzone porcje danych są następnie poddawane normalizacji.
116 def prepare_data(chunk_size_in_seconds):
117     input_data, target, headers = load_input_data(chunk_size_in_seconds)
118
119     chunks_input, chunks_target = prepare_chunks(input_data,
120                                                  target,
121                                                  chunk_size_in_seconds=chunk_size_in_seconds,
122                                                  ratio=FREQUENCY_TO_SAMPLING_RATIO)
123     x, y = flatten_chunks(chunks_input, chunks_target)
124     x, y = normalize(x, y)
125
126     return x, y
127
128
129 # Główna funkcja skryptu, która zwraca odpowiednio przygotowane dane. Przy pierwszym
        uruchomieniu wczytuje dane z dysku do zmiennych i zapisuje je do pliku
        tymczasowego. Odczyt zmiennych z danymi z pliku binarnego umożliwia szybszy dostęp
        przy kolejnych uruchomieniach, gdyż otwieranie dużych plików tekstowych jest
        czasochłonne.
130 def get_data(chunk_size_in_seconds):
131     file_exists = os.path.isfile(INPUT_DATA_FILE_PATH.format(chunk_size_in_seconds))
132
133     if not file_exists:
134         load_data_to_file(chunk_size_in_seconds)
135
136     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 zawieracjące 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 proporcjonalną 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 odopowiadającego za przetwarzanie danych do postaci okien czasowych przedstawiona została na listlingu 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         # atak oznaczony wartością '1'
18         chunks_target[seizure_number] = 1
19
```

```
20      return (chunks_input, chunks_target)
21
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
        sprawdzane jest, aby nie zawierała ona momentów, w których wystąpił atak.
35  def create_non_seizure_data_start_index(data_size, chunk_size, seizure_seconds):
36      start_index = random.randint(0, data_size - chunk_size)
37
38      while (is_in_seizure_range(start_index, seizure_seconds, chunk_size)):
39          start_index = random.randint(0, data_size - chunk_size)
40
41      return start_index
42
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,
                seizure_seconds)
54
55          chunk_end_index = chunk_start_index + chunk_size
56          chunks_input[chunk_number] = patient_data[chunk_start_index:chunk_end_index, :]
57          # brak ataku oznaczone wartością '0'
58          chunks_target[chunk_number] = 0
59
60      return (chunks_input, chunks_target)
61
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
        wielowymiarowej tablicy oraz tablicę zawierającą informacje o wystąpieniu ataków.
84  def prepare_chunks(input, target, chunk_size_in_seconds, ratio):
85      chunk_size = chunk_size_in_seconds * ratio
86      chunks_input = []
87      chunks_target = []
88
89      for patient_number in range(0, len(input)):
```

```
90          patient_chunks_input = []
91          patient_chunks_target = []
92          seizure_seconds = target[patient_number]
93          patient_data = input[patient_number]
94          (seizure_chunks_input, seizure_chunks_target) =
                create_chunks_with_seizures(patient_data,seizure_seconds, chunk_size)
95          patient_chunks_input.extend(seizure_chunks_input)
96          patient_chunks_target.extend(seizure_chunks_target)
97
98          (non_seizure_chunks_input, non_seizure_chunks_target) =
                create_chunks_without_seizures(patient_data, seizure_seconds, chunk_size)
99          patient_chunks_input.extend(non_seizure_chunks_input)
100         patient_chunks_target.extend(non_seizure_chunks_target)
101
102         chunks_input.append(np.array(patient_chunks_input))
103         chunks_target.append(np.array(patient_chunks_target))
104
105     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.

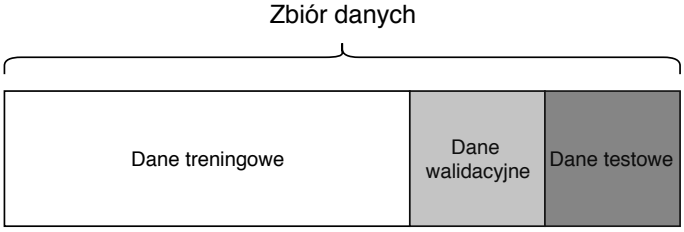
5.1.2. Metoda oceny wyników

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

5.1.2.1. 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 [2]. 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 5.1



Rys. 5.1. Podział danych na zbiory: treningowy, walidacyjny i testowy

Metoda ta rozwiązuje problem zbytniego dopasowywania się do zbioru testowego, jednak również posiada drugą 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.

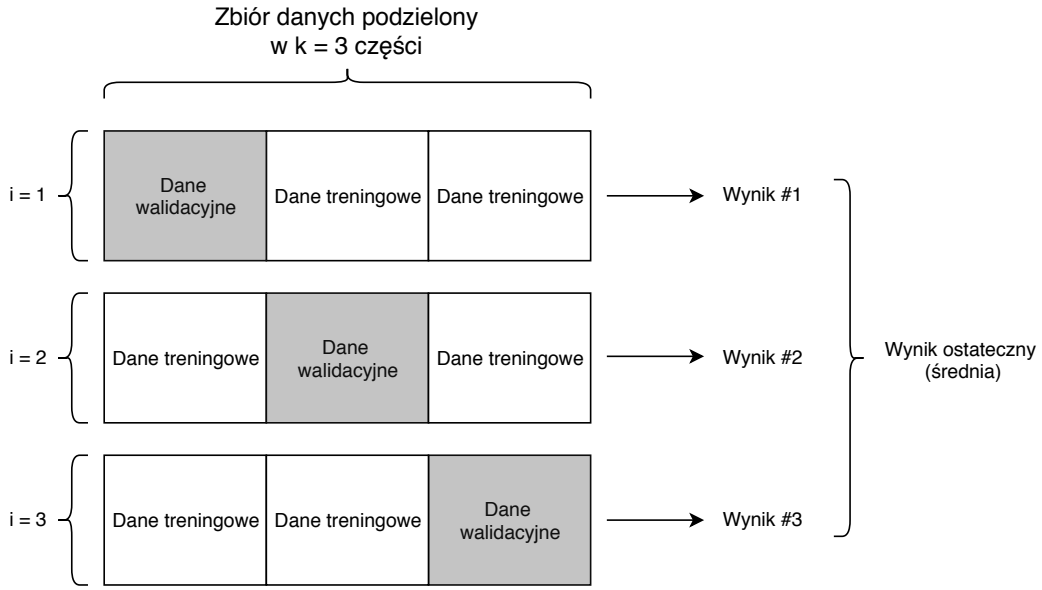
5.1.2.2. Walidacja krzyżowa

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 brana jest pod uwagę średnia skuteczność wszystkich modeli. Sposób podziału danych w tej metodzie ilustruje rysunek 5.2.

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.

5.1.2.3. Implementacja walidacji

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



Rys. 5.2. Podział danych według metody k-krotnej walidacji krzyżowe

Listing 5.5. Podział danych na zbiory

```
from sklearn.model_selection import train_test_split, StratifiedKFold
from data_reader import get_data

def load_data_kfold(folds_number, test_size=0.05)):
    x, y = get_data(CHUNK_SIZE_IN_SECONDS)

    x_train, x_test, y_train, y_test = train_test_split(x,
                                                        y,
                                                        test_size=test_size)

    folds = list(StratifiedKFold(n_splits=folds_number,
                                 shuffle=True,
                                 random_state=1).split(x_train, y_train))

    return folds, x_train, y_train, x_test, y_test

def load_data_train_test(test_size=0.05):
    x, y = get_data(CHUNK_SIZE_IN_SECONDS)

    x_train, x_test, y_train, y_test = train_test_split(x,
                                                        y,
                                                        test_size=test_size)

    return x_train, y_train, x_test, y_test
```

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. Iterecyjne wywołanie procesu uczenia

```
number_of_iterations = 5

avg_accuracies = []
std_accuracies = []

for iteration in range(0, number_of_iterations):
    iteration_number = iteration + 1
    print("Iteration", iteration_number)
```

```
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 5.1.3.

5.1.3. Uruchomienie uczenia właściwego

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

- tworzy model na podstawie dostarczonych specyfikacji (5.1.3.1)
- pobiera odpowiednie porcje danych dla każdego kroku z przytogowanych wcześniej zbiorów (5.1.2.3)
- tworzy listę tzw. *callback*'ów (5.1.3.2)
- uruchamia na modelu metodę służącą do dopasowania, która rozpoczyna uczenie (5.1.3.3)
- wczytuje wagi najlepszego modelu i sprawdza jego skuteczność (5.1.3.4)

Całe ciało metody *run_pipeline* zaprezentowane zostało 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          # rozpoczęcie uczenia
21          history = model.fit(x_train_cv,
22                              y_train_cv,
```

```
                            epochs=epochs,
                            batch_size=16,
                            callbacks=callbacks,
                            validation_data=(x_valid_cv, y_valid_cv),
                            verbose=0)

        # wczytanie wag najlepszego modelu i sprawdzenie jego skuteczności
        model.load_weights("tmp/best_model.h5")
        best_model_score.append(model.evaluate(x_valid_cv, y_valid_cv, batch_size=16,
            verbose=0))
        print("--Best model validation accuracy: %.2f%%" %
            (best_model_score[fold_number][1]*100))

    return best_model_score
```

5.1.3.1. Budowa modelu

Model tworzony jest na podstawie dostarczonych specyfikacji w postaci funkcji. Funkcja zawiera 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

```
def conv_1D_with_adam(input_shape):
    # opis modelu tworzony na podstawie nazwy funkcji
    description = get_function_name()

    # specyfikacja modelu tworzonego sekwencyjnie
    model = Sequential()

    # dodanie warstwy splotowej z 32 1-wymiarowymi filtrami o rozmiarze 6 z funkcją
        aktywacji 'relu'
    model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu',
        input_shape=input_shape))
    # warstwa max pooling'u o rozmiarze 2
    model.add(MaxPooling1D(pool_size=2))

    # warstwa zmniejszająca liczbę wymiarów danych
    model.add(Flatten())
    # warstwa typu 'fully-connected' o rozmiarze 64 neuronów
    model.add(Dense(64, activation='relu'))
    # warstwa typu 'fully-connected' z jednym neuronem i sigmoidalną funkcją aktywacji,
        która na wyjściu zwróci wartość '0' lub '1'
    model.add(Dense(1, activation='sigmoid'))

    # kompilacja modelu z użyciem optymalizatora Adam, funkcji strat binarnej entropii
        krzyżowej oraz dokładności (accuracy) jako metryki
    model.compile(optimizer=Adam(),
                  loss='binary_crossentropy',
                  metrics=['acc'])

    return model, description
```

5.1.3.2. 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 5.2

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

Listing 5.9. Tworzenie listy *callback*'ów

```
def step_decay(epoch):
    initial_lrate=0.1
    drop=0.6
    epochs_drop = 10.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))

    return lrate


def callbacks_list(description):
    return [
    callbacks.EarlyStopping(
        monitor='val_acc',
        patience=30
    ),
    callbacks.LearningRateScheduler(step_decay),

    callbacks.ModelCheckpoint(
        filepath='tmp/best_model.h5',
        monitor='val_acc',
        save_best_only=True
    ),
    callbacks.TensorBoard(
        log_dir='tmp/logs/{}. {}'.format(description, create_current_time()),
        histogram_freq=0,
        write_graph=True,
        write_images=True
    )
]
```

5.1.3.3. 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 5.2. Dodatkowo dla każdej

iteracji zapisywany jest najlepszy model, który na sam koniec zostanie załadowany i sprawdzony pod kątem skuteczności 5.1.3.4.

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

```
history = model.fit(x_train_cv,
                    y_train_cv,
                    epochs=epochs,
                    batch_size=16,
                    callbacks=callbacks,
                    validation_data=(x_valid_cv, y_valid_cv),
                    verbose=0)
```

5.1.3.4. 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

```
model.load_weights("tmp/best_model.h5")
best_model_score.append(model.evaluate(x_valid_cv, y_valid_cv, batch_size=16,
    verbose=0))

print("--Best model validation accuracy: %.2f%%" %
    (best_model_score[fold_number][1]*100))
```

5.2. Monitorowanie

Model sieci neuronowej po zakończeniu procesu nauki zwraca obiekt zawierający raport z wynikami. W sytuacji gdy proces uczenia przeprowadzany jest kilkukrotnie 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.

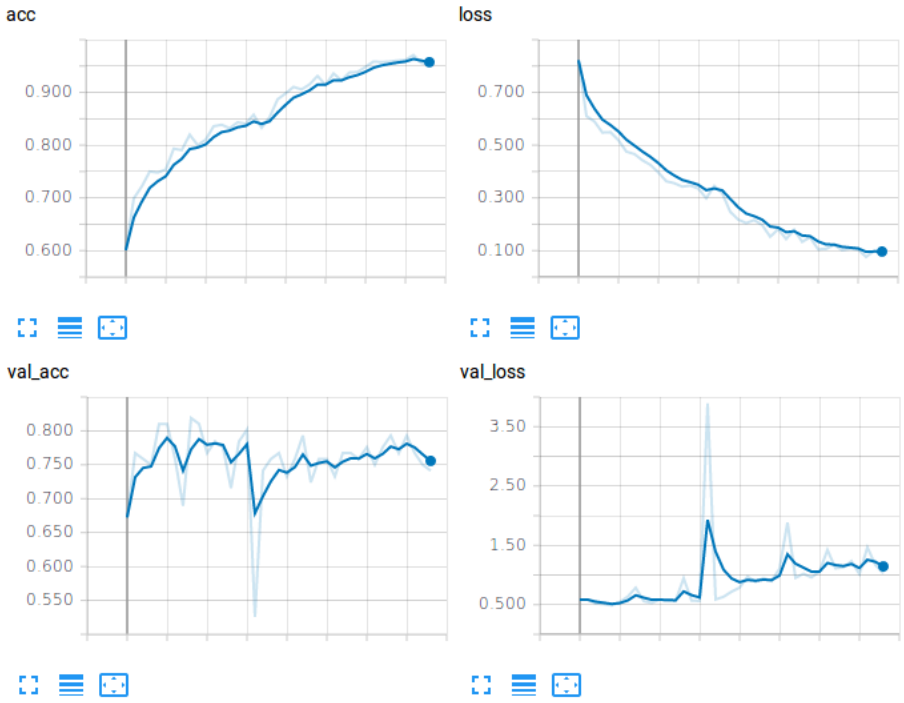
Do monitorowania procesu uczenia zostało użyte narzędzie *TensorBoard*, które dostarczone jest razem z biblioteką *TensorFlow*. Pozwala ono m.in. na graficzną wizualizację postępów w procesie uczenia oraz automatyczne generowanie schematu modelu sieci neuronowej. Jest to narzędzie uruchamiane w przeglądarce, które wykorzystuje logi wygenerowane przez *callback.TensorBoard* omawiany w rozdziale 5.1.3.2.

Na najprostszym widoku widoczne 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

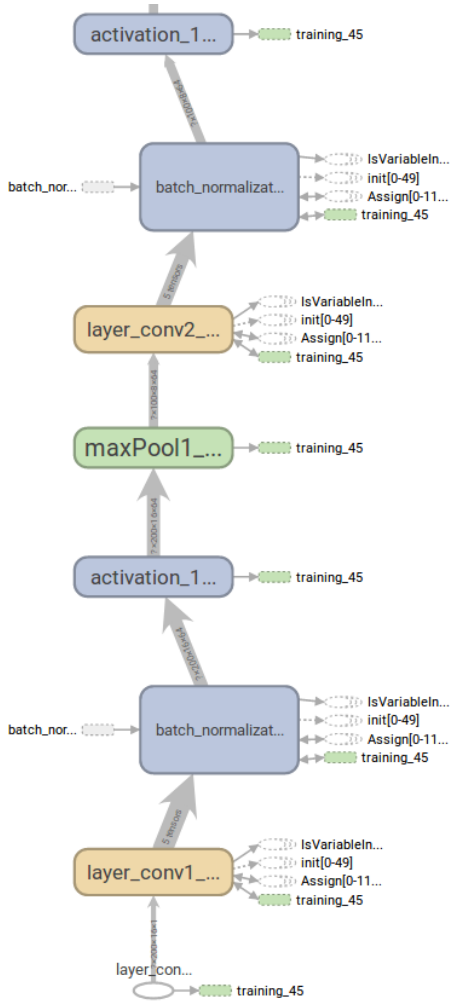
Wykresy zostały zaprezentowane na rysunku 5.3. Są one interaktywe, więc można sprawdzić dokładną wartość w każdym punkcie wykresu.



Rys. 5.3. Dynamicznie tworzone wykresy w programie TensorBoard

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 5.4.

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. 5.4. Fragment schematu modelu wygenerowany przez TensorBoard

5.3. 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 5.4.

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 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

5.3.1. Klasyczna sieć neuronowa typu fully connected

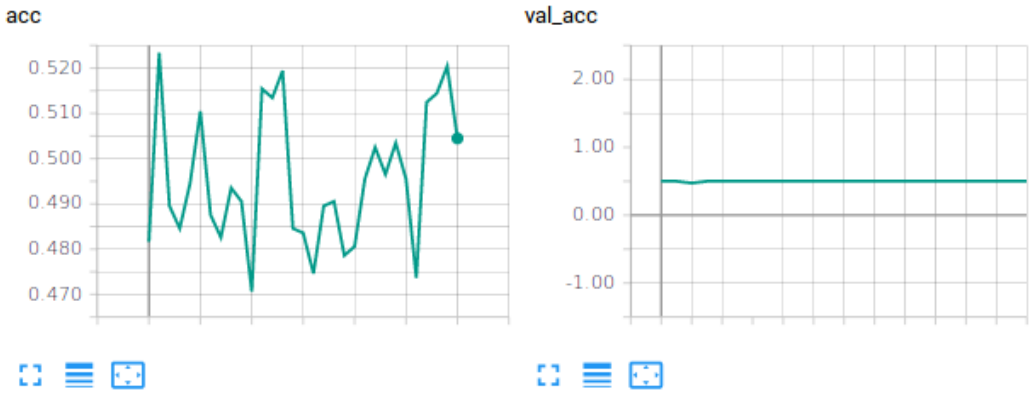
Pierwszym, najprostszym rozwiązaniem, które zostało przetestowane jest klasyczna sieć neuronowa typu *fully connected*. W bibliotece Keras warstwy tworzące taką sieć noszą nazwę *Dense*. Model początkowy, który został stworzony zaprezentowano na listingu 5.12.

Listing 5.12. Model sieci neuronowej typu *fully connected*

```
model = Sequential()
model.add(Flatten(input_shape=input_shape))
model.add(Dense(1000, kernel_initializer='normal', activation='relu'))
model.add(Dense(30, kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))

sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
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 tracone 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)*.

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%** 5.5.



Rys. 5.5. 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 warto-

ści. Pomimo próby modyfikacji sieci poprzez zwiększenie/zmniejsznie 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.

5.3.2. 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)*. Filtr jednowymiarowy 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 jednocześnie. Oznacza to, że być może sieć tego typu nie będzie w stanie zauważyć zależności pomiędzy danymi znajdującymi się na różnych kanałach.

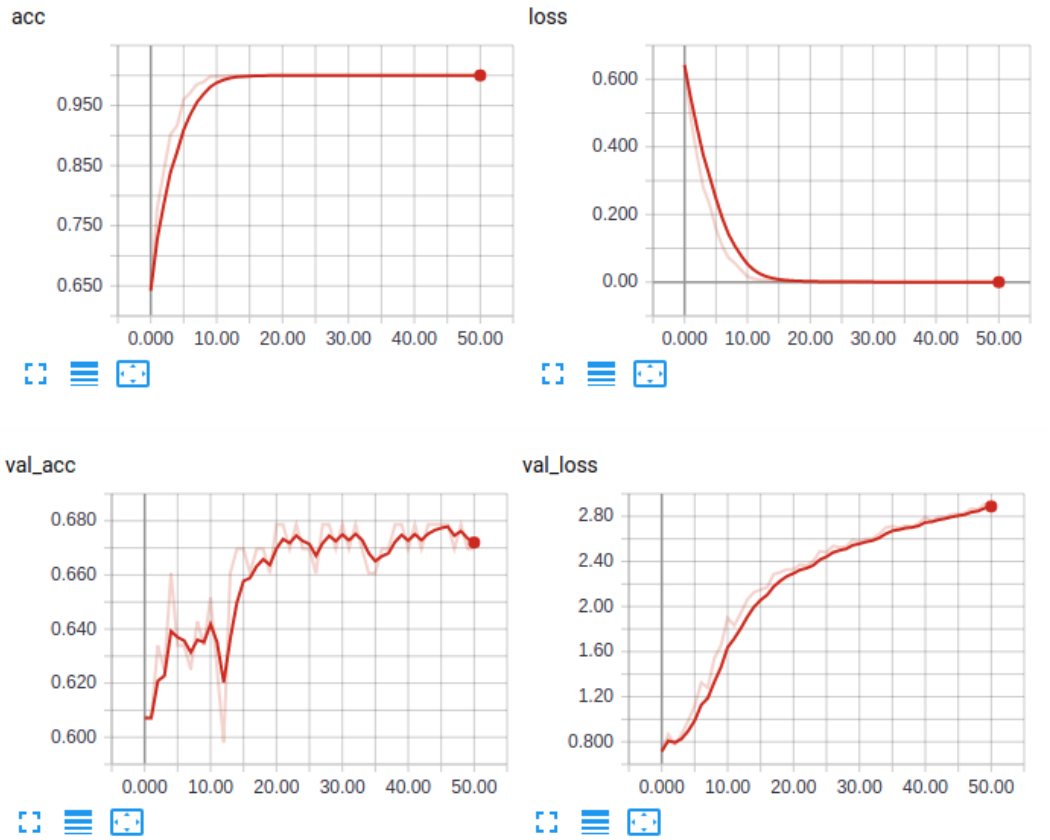
Wstępny model sieci, który został wybrany przedstawiony został na listingu 5.13. Składa się on z dwóch warstw konwolucyjnych o odpowiednio 32 i 16 filtrach z warstwami *max pooling*'u. Sieci splotowe zwykle zakończone są kilkoma warstwami typu *fully-connected*. W tym wypadku została zastosowana jedna 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.

Listing 5.13. Model splotowej sieci neuronowej z 1-wymiarowym filtrem

```
model = Sequential()

model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu',
    input_shape=input_shape))
model.add(MaxPooling1D(pool_size=2))

model.add(Conv1D(filters=16, kernel_size=6, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])
```

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)* 5.6.

Wyniki były obiecujące, więc sieć tego typu została wzięta pod uwagę w dalszych pracach polegających na optymalizacji w celu uzyskania jak najlepszych wyników 5.4.



Rys. 5.6. Wyniki prostej sieci splotowej z 1-wymiarowym filtrem

5.3.3. Sieć splotowa z filtrem 2-wymiarowym

Następnym typem sieci, która została przetestowana była sieć splotowa z filtrem 2-wymiarowym *(Conv2D)*. Sieć ta jest bardzo podobna do konwolucyjnej sieci jednowymiarowej opisywanej w poprzednim podrozdziale 5.3.2. 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.

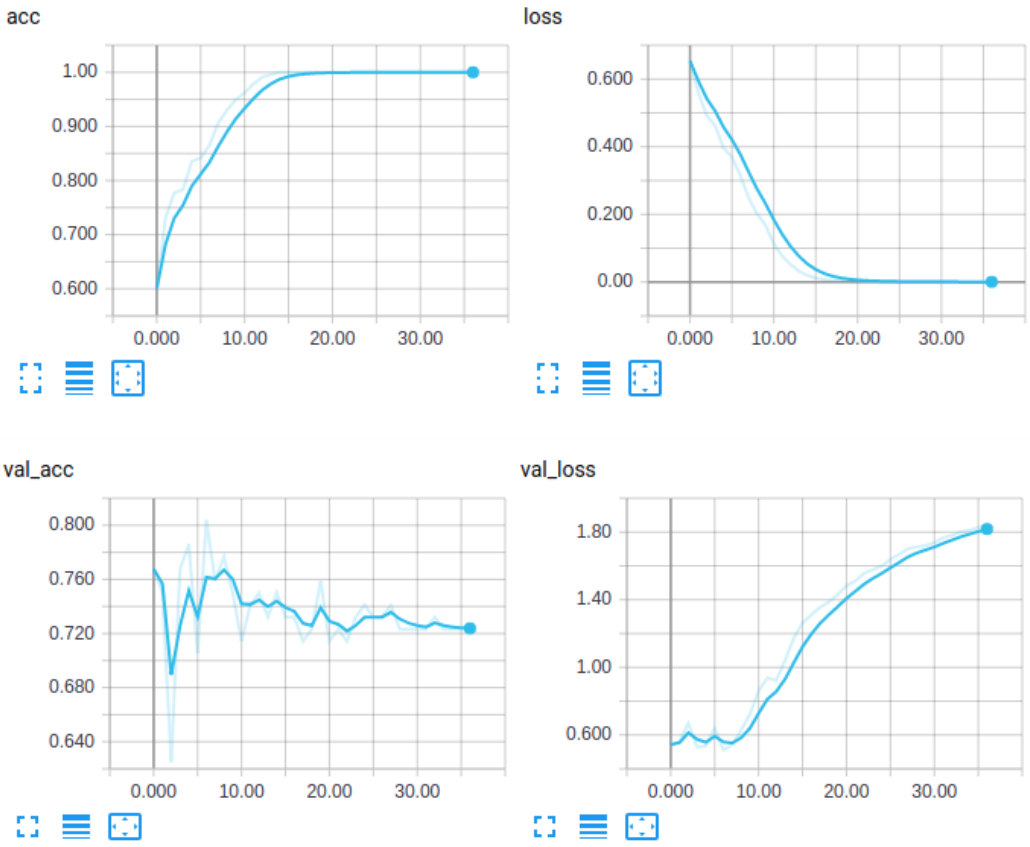
Analogicznie do modelu sieci z filtrem jednowymiarowym został stworzony podobny z 2-wymiarowym filtrem 5.14

Listing 5.14. Model splotowej sieci neuronowej z 2-wymiarowym filtrem

```
1  model = Sequential()
2
3  model.add(Conv2D(32,(3,3),strides = (1,1),padding='same', activation = 'relu',
       input_shape=input_shape))
4  model.add(MaxPooling2D((2,2)))
5
6  model.add(Conv2D(16,(3,3),strides = (1,1),name='conv3', activation = 'relu'))
7  model.add(MaxPooling2D((2,2)))
8
9  model.add(Flatten())
10 model.add(Dense(64,activation = 'relu'))
```

```
11 model.add(Dense(1, activation='sigmoid'))
12
13 model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['acc'])
```

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 zaprezentowanych na rysunku 5.7 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 5.4.



Rys. 5.7. Wyniki prostej sieci splotowej z 2-wymiarowym filtrem

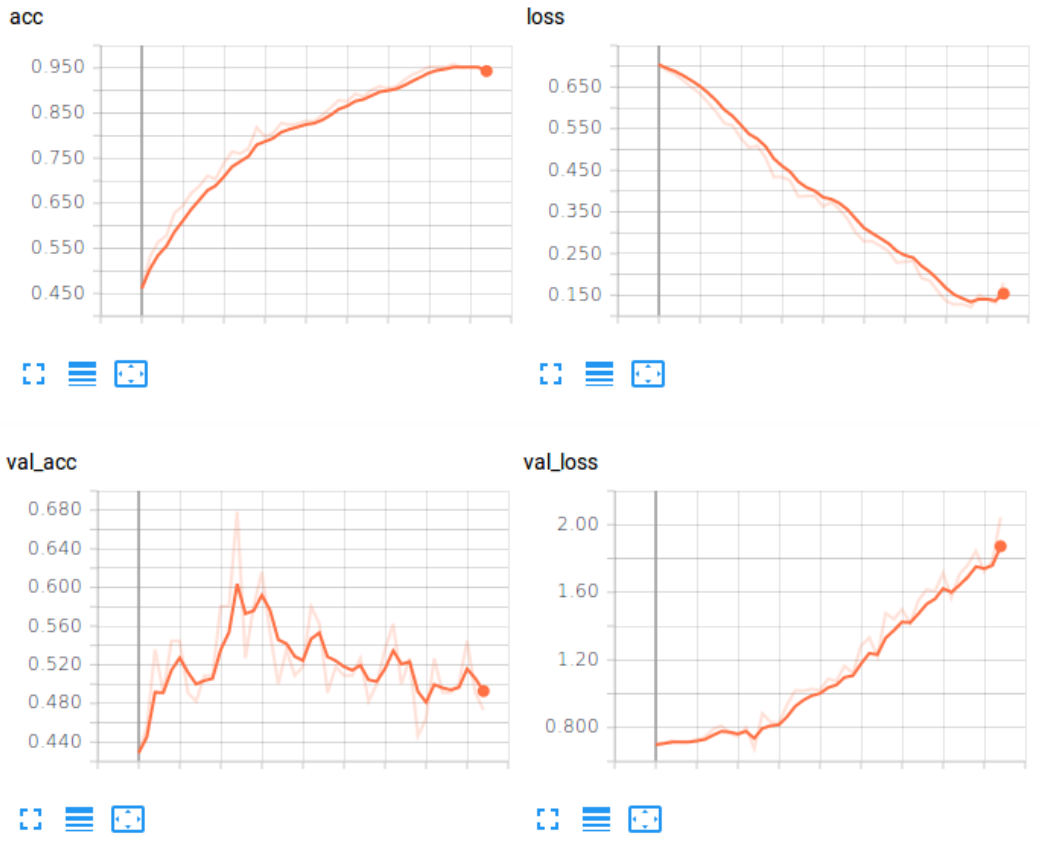
5.3.4. Sieć rekurencyjna

W następnym kroku przetestowana została rekurencyjna sieć neuronowa. Początkowy model sieci, który został utworzony zaprezentowany został na listingu 5.15.

Listing 5.15. 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'])
```

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%** 5.8.



Rys. 5.8. Wyniki prostej sieci rekurencyjnej

Jest to wynik o wiele gorszy w porównaniu do sieci konwolucyjnych testowanych w poprzednich rozdziałach. Próby rozbudowy i prostej optymalizacji sieci nie przynosiły 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 5.3.5

5.3.5. 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 stuktura sieci polegająca na użyciu kilku warstw konwolucyjnych zakończonych warstwami rekurencyjnymi. Prosty model sieci zbudowany według tej koncepcji zaprezentowany został na listingu 5.16

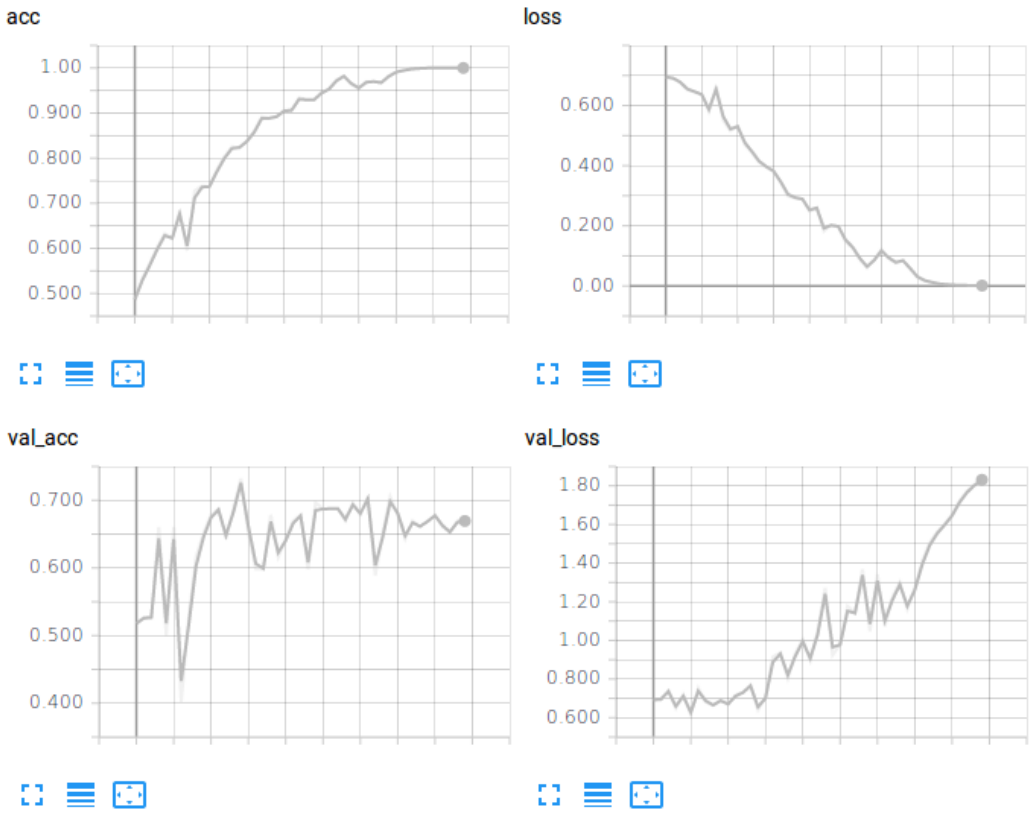
Listing 5.16. Model sieci typu CNN LSTM

```
1 model = Sequential()
```

```
2
3  model.add(Conv1D(filters=64, kernel_size=6, padding='same', activation='relu',
       input_shape=input_shape))
4  model.add(MaxPooling1D(pool_size=2))
5  model.add(Conv1D(filters=32, kernel_size=6, padding='same', activation='relu'))
6  model.add(MaxPooling1D(pool_size=2))
7
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 wynosił 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%** 5.9.



Rys. 5.9. Wyniki sieci konwoluncyjnej połączonej z rekurencyjną

Zostały podjęte próby roszerzenia 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.

5.3.6. 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 splotowa z filtrem 2-wymiarowym. Model tej sieci został więc wybrany jako ten najlepiej przystosowany do rozwiązywania badanego problemu. Próba i sposoby optymalizacji wyników tego modelu zostały opisane w kolejnym rozdziale 5.4.

5.4. Optymalizacja

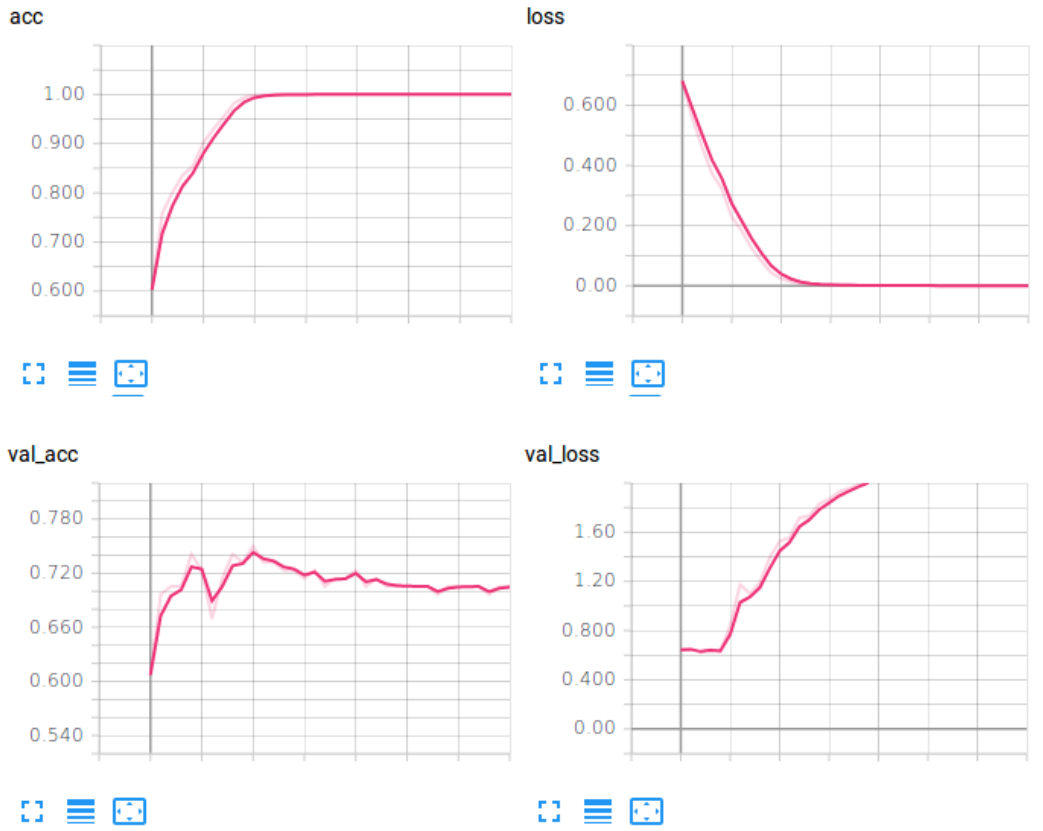
Sieć typu konwolucyjnego z filtrem 2-wymiarowym osiągała najlepsze wyniki testując prototypy różnych typów sieci neuronowych. To właśnie tej architekturze zostało więc poświęcone najwięcej pracy mającej na celu optymalizację i polepszenie otrzymywanych wyników. Wyniki już nawet przy zaprezentowanym prostym 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 zbytnim 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 5.10.

Istnieje kilka sposobów przeciwdziałania nadmiernego dopasowania. Zostaną one opisane w kolejnych rozdziałach.

5.4.1. 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



Rys. 5.10. Wyniki modelu początkowego konwolucyjnej sieci neuronowej

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].

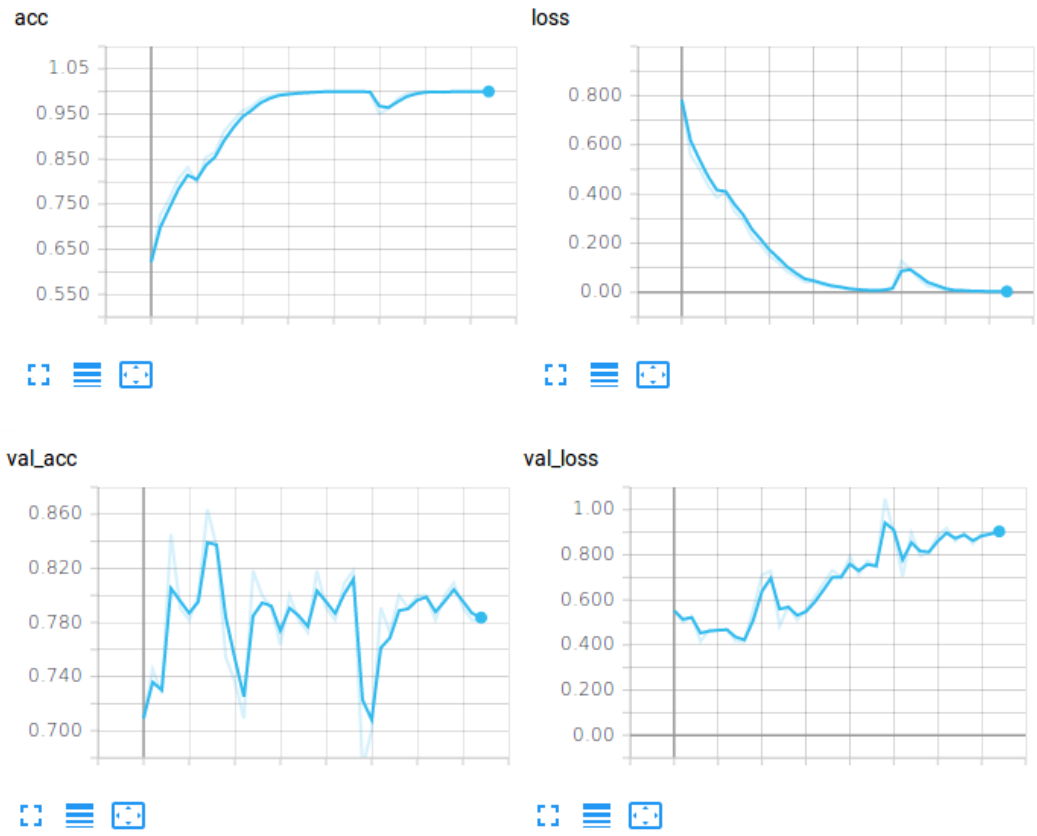
W bibliotece *Keras* dostępne są gotowe warstwy o nazwie *BatchNormalization* dostarczające opisaną funkcjonalność. Wystarczy dodać warstwę poprzedzającą funkcję aktywacji 5.17.

Listing 5.17. Model sieci konwolucyjnej z warstwami BatchNormalization

```
model = Sequential()

model.add(Conv2D(32, (3,3), strides = (1,1), padding='same', input_shape=input_shape))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(16, (3,3), strides = (1,1), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2)))

model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(1, activation='sigmoid'))

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 5.11



Rys. 5.11. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami *BatchNormalization*

5.4.2. Dropout

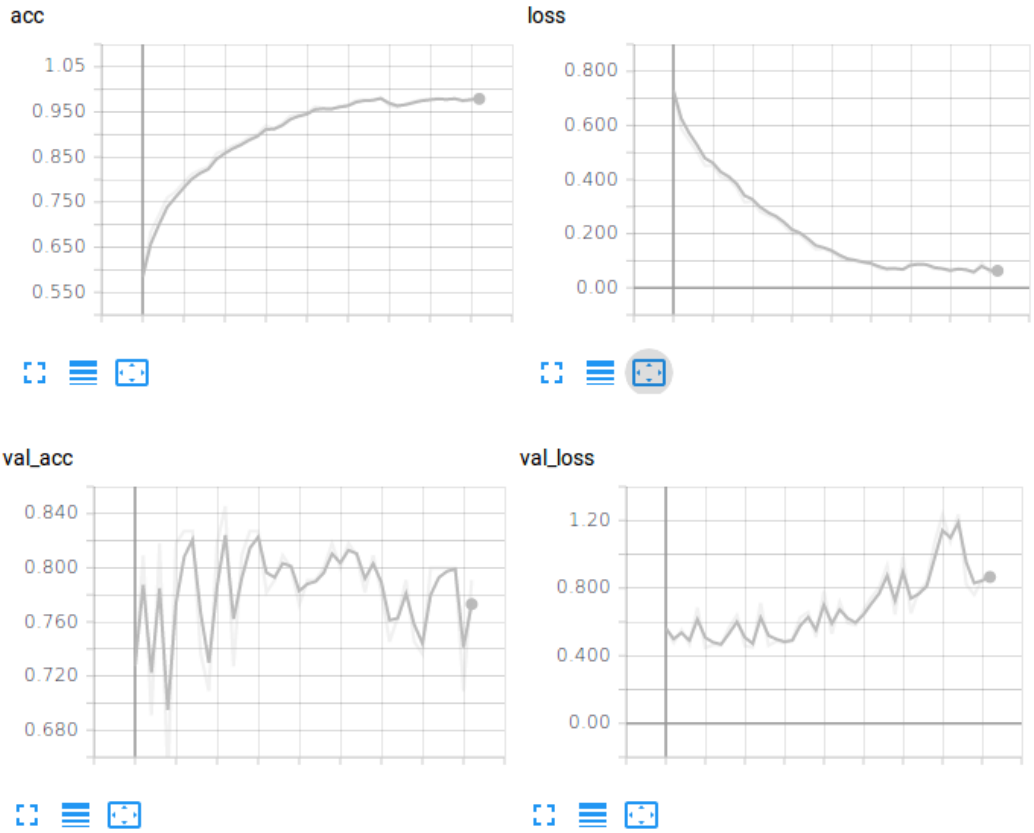
Kolejną techniką jest wykorzystanie tzw. *dropout*'u. Polega ona na tym, że wartości na warstwie wyjściowej są zerowane ze wskazanym prawdopodobieństwem. Dzięki temu podczas każdej epoki uczącej dane wyglądają trochę inaczej, niż poprzednio, dlatego sieć nie uczy się zawsze na podstawie tych samych danych, tylko ich drobnych modyfikacjach. W bibliotece *Keras* została zaimplementowana gotowa warstwa *Dropout*, która umożliwia wykorzystanie tej techniki 5.18. Została ona nałożona na wyniki warstwy typu *Dense*.

Listing 5.18. Model sieci konwolucyjnej z warstwami Dropout

```
model = Sequential()

model.add(Conv2D(32, (3,3), strides = (1,1), padding='same', input_shape=input_shape))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(16, (3,3), strides = (1,1), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
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 osiągana już skuteczność 100% oraz wartości funkcji strat równej 0. Podczas walidacji wartość funkcji strat dalej rośnie, jednak w mniejszym stopniu, niż poprzednio 5.12.



Rys. 5.12. Wyniki modelu konwolucyjnej sieci neuronowej z warstwami *Dropout*

5.4.3. 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 optymalizator tego typu bardzo dobrze sprawdzał się również w parze z adaptacyjnie dobieranym współczynnikiem uczenia 5.4.4.

Tab. 5.2. Porównanie skuteczności sieci przy użyciu poszczególnych optymalizatorów

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

5.4.4. 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 5.19 przedstawia regułę, według której będzie on modyfikowany.

Listing 5.19. 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
```

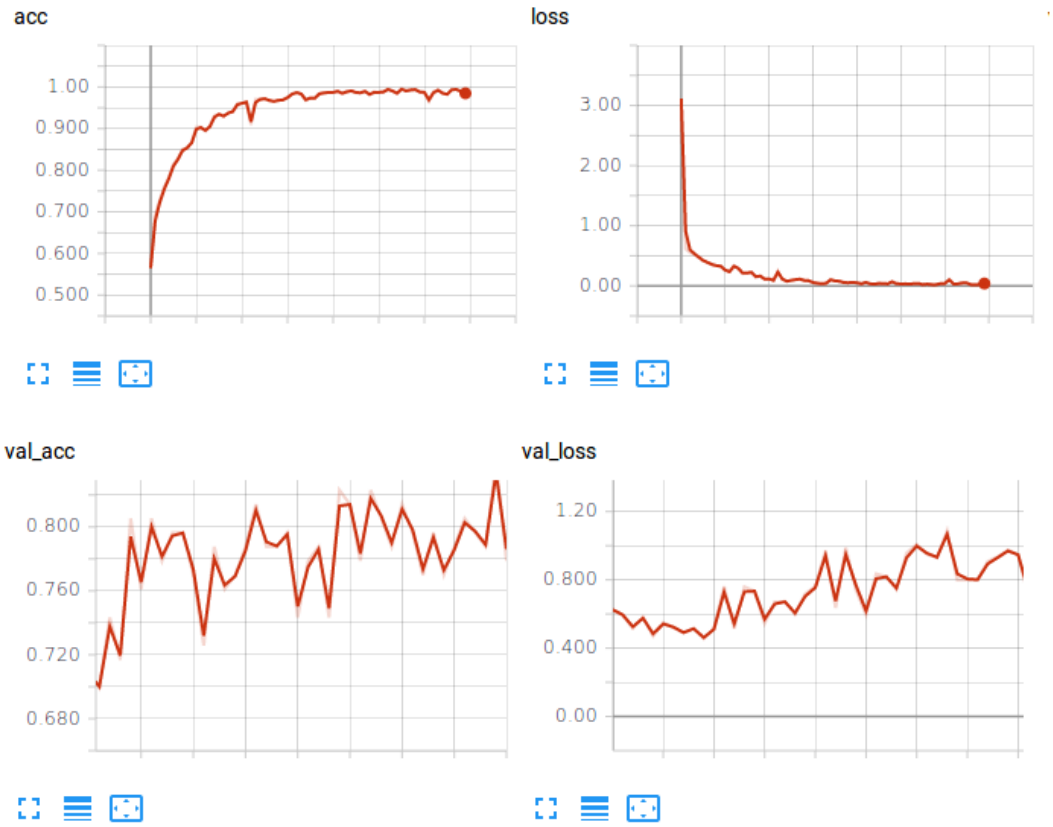
Dynamiczna zmiana współczynnika uczenia pozwoliła na częściowe zredukowanie nadmiernego dopasowania sieci, jednak wciąż zaobserwować można jej przeuczanie. Mimo to skuteczność została poprawiona do blisko **80,5%** 5.13.

5.4.5. 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 5.20 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*.

Podsumowanie modelu wygenerowane za pomocą funkcji *model.summary()* pokazujące użyte warstwy, rozmiary wyjściowe oraz ilość parametrów przedstawia się następująco 5.21.



Rys. 5.13. Wyniki modelu po dodaniu zmiennego współczynnika uczenia

Listing 5.20. Ostateczny model sieci konwolucyjnej

```
model = Sequential()

model.add(Conv2D(64,(3,3),strides = (1,1),name='layer_conv1',padding='same',
    input_shape=input_shape))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2),name='maxPool1'))

model.add(Conv2D(64,(3,3),strides = (1,1),name='layer_conv2',padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2),name='maxPool2'))

model.add(Conv2D(32,(3,3),strides = (1,1),name='conv3',padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2),name='maxPool3'))

model.add(Flatten())
model.add(Dense(64,activation = 'relu',name='fc0'))
model.add(Dropout(0.25))
model.add(Dense(32,activation = 'relu',name='fc1'))
model.add(Dropout(0.25))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=SGD(lr=0.01, momentum=0.5, decay=0.0, nesterov=False),
              loss='binary_crossentropy',
              metrics=['acc'])
```

Listing 5.21. Podsumowanie modelu

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
layer_conv1 (Conv2D)         (None, 200, 16, 64)       640
_________________________________________________________________
batch_normalization_88 (Batc (None, 200, 16, 64)       256
_________________________________________________________________
activation_88 (Activation)   (None, 200, 16, 64)       0
_________________________________________________________________
maxPool1 (MaxPooling2D)      (None, 100, 8, 64)        0
_________________________________________________________________
layer_conv2 (Conv2D)         (None, 100, 8, 64)        36928
_________________________________________________________________
batch_normalization_89 (Batc (None, 100, 8, 64)        256
_________________________________________________________________
activation_89 (Activation)   (None, 100, 8, 64)        0
_________________________________________________________________
maxPool2 (MaxPooling2D)      (None, 50, 4, 64)         0
_________________________________________________________________
conv3 (Conv2D)               (None, 50, 4, 32)         18464
_________________________________________________________________
batch_normalization_90 (Batc (None, 50, 4, 32)         128
_________________________________________________________________
activation_90 (Activation)   (None, 50, 4, 32)         0
_________________________________________________________________
maxPool3 (MaxPooling2D)      (None, 25, 2, 32)         0
_________________________________________________________________
flatten_30 (Flatten)         (None, 1600)              0
_________________________________________________________________
fc0 (Dense)                  (None, 64)                102464
_________________________________________________________________
dropout_59 (Dropout)         (None, 64)                0
_________________________________________________________________
fc1 (Dense)                  (None, 32)                2080
_________________________________________________________________
dropout_60 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_30 (Dense)             (None, 1)                 33
=================================================================
Total params: 161,249
Trainable params: 160,929
Non-trainable params: 320
```

Na listingu 5.22 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 5.22. Wyniki skuteczności

```
Iteration 1

Fold:  0
--Best model validation accuracy: 79.65%

Fold:  1
--Best model validation accuracy: 83.93%

Fold:  2
--Best model validation accuracy: 83.04%

Fold:  3
--Best model validation accuracy: 84.82%

Fold:  4
--Best model validation accuracy: 86.61%

Fold:  5
--Best model validation accuracy: 77.68%

Fold:  6
```

```
--Best model validation accuracy: 81.08%

Fold:  7
--Best model validation accuracy: 78.38%

Fold:  8
--Best model validation accuracy: 81.98%

Fold:  9
--Best model validation accuracy: 83.78%


Best models average validation accuracy: 0.820943
Best models standard deviation of accuracy: 0.027431
Iteration 1 time: 511.17 seconds


Iteration 2

Fold:  0
--Best model validation accuracy: 78.76%

Fold:  1
--Best model validation accuracy: 83.04%

Fold:  2
--Best model validation accuracy: 83.93%

Fold:  3
--Best model validation accuracy: 84.82%

Fold:  4
--Best model validation accuracy: 87.50%

Fold:  5
--Best model validation accuracy: 75.89%

Fold:  6
--Best model validation accuracy: 77.48%

Fold:  7
--Best model validation accuracy: 83.78%

Fold:  8
--Best model validation accuracy: 78.38%

Fold:  9
--Best model validation accuracy: 84.68%


Best models average validation accuracy: 0.818264
Best models standard deviation of accuracy: 0.036667
Iteration 2 time: 527.52 seconds


Iteration 3

Fold:  0
--Best model validation accuracy: 81.42%

Fold:  1
--Best model validation accuracy: 85.71%

Fold:  2
--Best model validation accuracy: 83.93%

Fold:  3
--Best model validation accuracy: 84.82%

Fold:  4
--Best model validation accuracy: 86.61%

Fold:  5
```

```
--Best model validation accuracy: 76.79%

Fold:  6
--Best model validation accuracy: 77.48%

Fold:  7
--Best model validation accuracy: 81.98%

Fold:  8
--Best model validation accuracy: 82.88%

Fold:  9
--Best model validation accuracy: 84.68%


Best models average validation accuracy: 0.8263
Best models standard deviation of accuracy: 0.031417
Iteration 3 time: 643.31 seconds


~~~Grand mean of average accuracy: 0.821836
~~~Grand mean of standard deviation accuracy: 0.031839
```

5.5. 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 naczonego 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 5.23 jest skuteczność modelu na danych, które zostały użyte po raz pierwszy 5.24.

Listing 5.23. Sprawdzanie skuteczności modelu na danych testowych

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

Listing 5.24. Skuteczność modelu na danych testowych

```
Iteration 1

Fold:  0
--Best model test accuracy: 77.97%

Fold:  1
--Best model test accuracy: 79.66%

Fold:  2
--Best model test accuracy: 79.66%

Fold:  3
--Best model test accuracy: 72.88%

```

```
Fold:  4
--Best model test accuracy: 79.66%

Fold:  5
--Best model test accuracy: 79.66%

Fold:  6
--Best model test accuracy: 74.58%

Fold:  7
--Best model test accuracy: 81.36%

Fold:  8
--Best model test accuracy: 76.27%

Fold:  9
--Best model test accuracy: 77.97%

Best models average test accuracy: 0.779661


Iteration 2

Fold:  0
--Best model test accuracy: 74.58%

Fold:  1
--Best model test accuracy: 79.66%

Fold:  2
--Best model test accuracy: 74.58%

Fold:  3
--Best model test accuracy: 74.58%

Fold:  4
--Best model test accuracy: 77.97%

Fold:  5
--Best model test accuracy: 72.88%

Fold:  6
--Best model test accuracy: 67.80%

Fold:  7
--Best model test accuracy: 71.19%

Fold:  8
--Best model test accuracy: 79.66%

Fold:  9
--Best model test accuracy: 74.58%

Best models average test accuracy: 0.747458


Iteration 3

Fold:  0
--Best model test accuracy: 79.66%

Fold:  1
--Best model test accuracy: 74.58%

Fold:  2
--Best model test accuracy: 76.27%

Fold:  3
--Best model test accuracy: 77.97%

Fold:  4
--Best model test accuracy: 79.66%

```

```
Fold:  5
--Best model test accuracy: 76.27%

Fold:  6
--Best model test accuracy: 72.88%

Fold:  7
--Best model test accuracy: 83.05%

Fold:  8
--Best model test accuracy: 77.97%

Fold:  9
--Best model test accuracy: 81.36%

Best models average test accuracy: 0.779661


~~~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%** (76.8927%) na danych testowych jest zadowalający i przekracza założony próg 70% 3.3.

5.6. Możliwości rozwoju

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

- Próba zwalczenia zjawiska nadmiernego dopasowania dla zaproponowanego modelu
- Przetworzenie danych do innej postaci (*ang. feature extraction*) np. wizualizacja aktywności mózgu [14] lub stworzenie tzw. *heat map*'y metodą *matching pursuit* [15]
- Wykorzystanie gotowych modeli sieci wbudowanych w bibliotekę *Keras* np. *Xception*, *ResNet*, *VGG19* itd. [16]

Dodatek A

Płyta DVD

Do tesktu 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:

  // todo: uzupełnić

  - katalog **/program/** - program,

Dodatek B

Użycie biblioteki Keras w języku R

Biblioteka Keras powszechnie używana jest pisząc kod w języku 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 5.4) w języku R i środowisku RStudio.

B.1. Implementacja przykładowego modelu do klasyfikacji danych MNIST

Po uruchomieniu środowiska RStudio należy zainstalować bibliotekę keras B.1.

Listing B.1. Instalacja Keras

```
devtools::install_github("rstudio/keras")

library(keras)
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 B.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 B.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 dodatkowo przeskalować, aby znalazły się one w przedziale <0;1> B.3.

Rys. B.1. Przykładowe dane ze zbioru MNIST

Listing B.2. Wczytywanie danych MNIST

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

Listing B.3. Przygotowanie danych wejściowych

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))

x_train <- x_train / 255
x_test <- x_test / 255
```

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

Listing B.4. Przygotowanie danych wyjściowych

```
y_train <- to_categorical(y_train, 10)
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 pradopowobieństwo zajścia jednego z 10 stanów B.5.

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

Listing B.6. Rozpoczęcie procesu uczenia

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 50, batch_size = 64,
  validation_split = 0.2
)
```

Listing B.5. Utworzenie modelu

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')

model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

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

Listing B.7. Sprawdzenie modelu na danych testowych

```
model %>% evaluate(x_test, y_test,verbose = 0)


$loss
[1] 0.1241

$acc
[1] 0.9788
```

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

Na listingu B.8 zaprezentowany został model konwolucyjnej sieci neuronowej utworzony w języku R odzwierciedlający sieć użytą do rozpoznawania stanów padaczkowych przedstawioną w rozdziale 5.4.

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

```
model <- keras_model_sequential() %>%

  layer_conv_2d(filters = 64, kernel_size = c(3, 3), input_shape = input_shape) %>%
  layer_batch_normalization() %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  layer_conv_2d(filters = 64, kernel_size = c(3, 3), input_shape = input_shape) %>%
  layer_batch_normalization() %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  layer_conv_2d(filters = 32, kernel_size = c(3, 3), input_shape = input_shape) %>%
  layer_batch_normalization() %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate=0.25) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dropout(rate=0.25) %>%

```

```
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')
```

Bibliografia

[1] engmrk.com. *Module 22 – Implementation of CNN Using Keras*. https://engmrk.com/module-22-implementation-of-cnn-using-keras/.

[2] Francois Chollet. *Deep Learning with Python*. 2018.

[3] Jacob M. Williams. *Deep Learning and Transfer Learning in the Classification of EEG Signals*. 2017.

[4] Marvin Minsky and Seymour Paper. *Perceptrons*. 1969.

[5] futureoflife.org. *Benefits & risks of artificial intelligence*. https://futureoflife.org/background/benefits-risks-of-artificial-intelligence.

[6] searchenterpriseai.techtarget.com. *Machine learning (ML)*. https://searchenterpriseai.techtarget.com/definition/machine-learning-ML.

[7] Yoshua Bengio Ian Goodfellow and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[8] Epilepsy foundation. *About Epilepsy: The Basics*. https://www.epilepsy.com/learn/about-epilepsy-basics.

[9] Andrew Lim Pierre Thodoroff, Joelle Pineau. *Learning Robust Features using Deep Learning for Automatic Seizure Detection*. 2016.

[10] 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*.

[11] Nick Hershey. *Detecting Epileptic Seizures in Electroencephalogram Data*.

[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] Andrew Lim Pierre Thodorof, Joelle Pineau. *Learning Robust Features using Deep Learning for Automatic Seizure Detection*. arXiv:1608.00220v1 [cs.LG] 31 Jul 2016.

[15] Rafał Scherer-Ryszard Tadeusiewicz Lotfi A. Zadeh Jacek M. Zurada (Eds.) Leszek Rutkowski, Marcin Korytkowski. *Artificial Intelligence and Soft Computing*. 2013.

[16] Keras. *Keras docs*. `https://keras.io/applications/`.