**Projekt Sztuczne Sieci Neuronowe**

Temat: Przewidywanie wyniku rozgrywek szachowych

Przedmiot: Elementy sztucznej inteligencji

Spis treści

[1. Opis 3](#_Toc104664008)

[1.1. Opis problemu 3](#_Toc104664009)

[1.2. Wprowadzanie teoretyczne 4](#_Toc104664010)

[2. Przegląd literatury 5](#_Toc104664011)

[3. Analiza 6](#_Toc104664012)

[3.1. Ładowanie i eksploracja danych 6](#_Toc104664013)

[3.2. Wybór i uczenie modelu 7](#_Toc104664014)

[3.3. Test modelu ze zmienioną zmienną objaśniającą 11](#_Toc104664015)

[3.4. Skuteczność modelu w zależności od zmiany liczby epoch i batch size 13](#_Toc104664016)

[3.5. Skuteczność modelu w zależności od liczby neuronów 14](#_Toc104664017)

[3.6. Skuteczność modelu w zależności od liczby warstw ukrytych 15](#_Toc104664018)

# 1. Opis

## 1.1. Opis problemu

Szachy są jedną z najbardziej znanych i szanowanych gier na świecie, a ich historia ma już prawie 1500 lat. Za kolebkę szachów uznawane są Indie. Według źródeł pisanych gra ta była już znana w Persji na dworze szacha Chosrowa I Anoszirwana w latach 70. VI wieku naszej ery, gdzie przywieziona została w darze od indyjskiego radży. Gra ta, wymagająca strategicznego i taktycznego myślenia, przewijała się przez wieki historii jako możliwość pojedynku dla wielkich umysłów. Stosunkowo prosta plansza i zasady gry oraz praktycznie nieskończona liczba możliwych scenariuszy gry, spowodowały że szachy zdobyły serca wszystkich klas społecznych.

Podjętym w projekcie problemem będzie próba przewidywanie wyniku spotkania szachowego, na podstawie danych o meczu. Próba taka zdecydowanie nie należy do najłatwiejszych. Pomimo możliwości jakie dają nam współczesne technologie, dostępna moc obliczeniowa, biblioteki sieci neuronowych, to liczba możliwych kombinacji w szachach jest ogromna. Przy założeniu, że dla każdego ruchu mamy około 30 możliwości i przeciętna partia szachowa składa się z 80 ruchów, to mamy aż 3080 możliwości. Liczba ta w przybliżeniu wynosi 10120 i nazwana została liczbą shanona. W naszym projekcie będziemy pracowali na zbiorze 10 000 obserwacji, każda z nich reprezentuje jedno spotkanie szachowe. Zbiór danych pochodzi ze strony kaggle.com i jest dostępny pod adresem: <https://www.kaggle.com/datasets/datasnaek/chess>. Dla każdego z meczy zebrane zostały następujące dane:

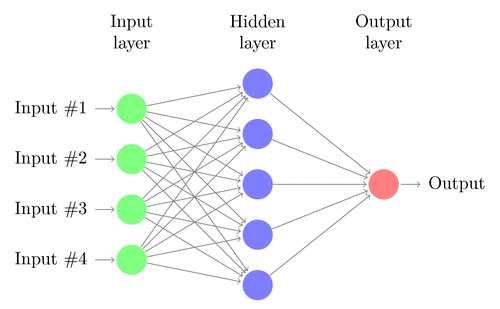
* id – numer identyfikujący spotkanie
* rated – czy mecz był rankingowy
* created\_at – czas rozpoczęcia gry
* last\_move\_at – czas ostatniego ruchu
* turns – sumaryczna liczba posunięć
* victory\_status – czy pojedynek został wygrany, poddany, zremisowany, zakończony przez upłynięcie czasu
* winner – czy mecz zakończony wygraną białego, czarnego czy remisem
* increment\_code – początkowy czas trwania spotkania, oraz o ile zwiększa się czas po każdym ruchu
* white\_id – id gracza poruszającego się białymi pionami
* white\_rating – ranking gracza poruszającego się białymi pionami
* black\_id – id gracza poruszającego się czarnymi pionami
* black\_rating - ranking gracza poruszającego się czarnymi pionami
* moves – zakodowane posunięcia
* opening\_eco – standardowy kod strategia rozpoczęcia
* opening\_name – nazwa strategii rozpoczęcia
* opening\_ply – liczba ruchów w fazie początkowej

## 1.2. Wprowadzanie teoretyczne

**Wprowadzenie teoretyczne**

Sztuczne sieci neuronowe (SSN) są systemem do przetwarzania informacji, który wzorowany jest na biologicznym systemie nerwowym. W sposób uproszczony odwzorowują działanie ludzkiego mózgu. SSN składają się z warstwy wejściowej oraz wyjściowej, pomiędzy którymi może znajdować się nieskończenie wiele warstw ukrytych. Każda z warstw składa się z neuronów, w przypadku warstwy wejściowej liczba neuronów odpowiada liczbie danych wejściowych, natomiast w warstwie wyjściowej pojawia się neuron, który wyświetla pożądaną informację. W najprostszym przypadku sieci neuronowej z jedną warstwą ukrytą, każdy neuron warstwy wejściowej połączony jest z każdym neuronem warstwy ukrytej, a neurony warstwy ukrytej połączone są z neuronem warstwy wyjściowej. Siła oddziaływania jednego neurony na drugi nazywana jest wagą i dobierana jest w taki sposób, by uzyskać jak najdokładniejszy wynik.[[1]](#footnote-1) Sumy iloczynów neuronów oraz wag tworzą blok sumujący, który przesyła wynik do funkcji aktywacji. Funkcja aktywacji przyjmuje postać liniową, nieliniową lub skoku jednostkowego. Jej wybór zależy od rodzaju problemu, który jest do rozwiązania.[[2]](#footnote-2)

Istnieją dwa podstawowe typy uczenia sieci neuronowych – uczenie z nauczycielem oraz uczenie bez nauczyciela. Uczenie z nauczycielem charakteryzuje się znajomością pożądanego wyniku. Wagi dopierane są w taki sposób, by wynik wyjścia sieci był zbliżony do wartości rzeczywistej. Z kolei uczenie bez nauczyciela polega na przekazywaniu do sieci jedynie danych wejściowych. Taki proces nazywany jest samouczeniem.[[3]](#footnote-3)



Rysunek 2 Budowa sztucznej sieci neuronowej z jedną warstwą ukrytą  
Źródło: https://research.aimultiple.com/how-neural-networks-work/

# 2. Przegląd literatury

Złożoność gry w szachy sprawia, że jest ona popularnym ,,celem” dla naukowców zajmujących się sztuczną inteligencją. W 1996 roku komputer firmy IBM „Deep Blue” jako pierwszy odniósł sukces, pokonując wówczas arcymistrza szachowego Garry’ego Kasparowa.

W literaturze częściej jednak znajdziemy próby przewidzenia kolejnych ruchów, na podstawie ruchów poprzednich, aniżeli tego, kto wygra cały pojedynek.

W artykule „End-to-End Deep Neural Network for Automatic Learning in Chess” [[4]](#footnote-4) autorstwa Konstantina Heruda używa on wstępnej klasyfikacji oraz podejścia regresywnego, aby ocenić możliwe konfiguracje gry poprzez zastosowanie sieci głębokiej wiary. Trzecie podejście badawcze, które łączy powyższe metody, jest następnie rozwijane poprzez kwantyzację zakresu wartości funkcji aktywacji. Sieć neuronowa uczy się dokładnie oceniać pozycje figur podczas nienadzorowanego treningu wstępnego oraz nadzorowanej fazy dostrajania modelu przy użyciu zbioru danych składającego się wyłącznie z wektora binarnego reprezentującego planszę oraz odpowiadających mu ocen. Algorytm alfa-beta jest używany jako uzupełnienie silnika szachowego w celu znalezienia optymalnych ruchów. Eksperyment pokazuje, jak sztuczne sieci neuronowe mogą rozwinąć ,,zrozumienie” dla badanego problemu, pomimo braku wcześniejszej wiedzy na temat zasad lub strategii gry.

Inną metodę wykorzystują Barak Oshri i Nishith Khandwala z Uniwersytetu Stanforda w artykule „Predicting Moves in Chess using Convolutional Neural Networks” [[5]](#footnote-5)używając trzywarstwowych konwolucyjnych sieci neuronowych do przewidywania ruchów pionów. Autorzy podzielili zadanie na dwie części: selektor figur CNN jest trenowany, aby określać, które białe pionki powinny się ruszyć, a selektor ruchów CNN określa, jaki ruch będzie najkorzystniejszy dla każdego z pionków. Takie działanie znacznie zredukowało złożoność obliczeniową modelu. Sieci zostały wytrenowane przy użyciu 245 000 ruchów składających się na 20 000 rozgrywek. Po uzyskaniu końcowych wyników autorzy zalecają używanie warstw konwolucyjnych do rozpoznawania wzorców raczej niewielkich taktyk i rekomendują przeszkolenie i ,,skomponowanie” tej taktyki z funkcjami aktywacji dla lepszej ogólnej gry.

# 3. Analiza

## 3.1. Ładowanie i eksploracja danych

Do przeprowadzenia analizy wykorzystamy biblioteki dostępne w języku Python. W głównej mierze będziemy posługiwać się bibliotekami pandas, numpy do obróbki danych, keras do budowy i testowania modelu sieci neuronowych oraz matplotlib do wizualizacji danych na wykresach. Po pobraniu danych przystępujemy do wstępnej obróbki danych, przygotowania ich do wstawienia do modelu.

Obraz zawierający tekst, monitor, zrzut ekranu, ekran

Opis wygenerowany automatycznie

Przy próbie zamiany czasu na typ datetime, napotkaliśmy problem, ponieważ data rozpoczęcia i zakończenia każdej obserwacji jest taka sama, zmienne te nie wnoszą zbyt wiele informacji. Ciężko będzie nam wyciągnąć z nich długość meczu, ponieważ różnica zawsze będzie wynosiła zero. Postanowiliśmy wyciągnąć z tej zmiennej godzinę rozpoczęcia meczu, gdyż uważamy że może ona mieć wpływ na wynik spotkania. Aby zapewnić sieci neuronowej lepsze możliwości nauki, zakodujemy godzinę jak parę dwóch zmiennych (sinusoidalnej oraz cosinusoidalnej) zgodnie z artykułem[[6]](#footnote-6) o przygotowywaniu zmiennych czasowych do sieci neuronowych.



Zakodowana godzina rozpoczęcia spotkania jest teraz reprezentowana przez kombinację dwóch zmiennych, zawierających się w zakresie (-1, 1).

W następnym kroku wyrzucimy więc zmienne *created\_at, last\_move\_at, victory\_status* ponieważ jest to tylko sposób w jaki zakończył się mecz, jeżeli będziemy przewidywać nie rodzaj a ogólny wynik meczu to nie będzie ona miała dla nas większego znaczenia. Wyrzucamy także zmienne *white\_id, black\_id, moves, opening\_name, id.*

Po przeprowadzeniu powyższych przekształceń otrzymujemy zbiór z następującymi kolumnami:

Obraz zawierający tekst, droga, zewnętrzne, monitor

Opis wygenerowany automatycznie

W kolejnym kroku musimy jeszcze zakodować zmienne *rated, winner, incremeant\_code oraz opening\_eco.* Zmienna *rated jest typu boolean,* więc wystarczy że zamienimy ją na zmienną zero-jedynkowa. Natomiast *winner, incremeant\_code, opening\_eco* są zmiennymi kategorycznymi z wieloma wartościami. Do zakodowania ich wykorzystamy funkcję get\_dummies() z pakietu pandas, która dla każdej wartości z danej zmiennej generuje kolumnę, w której jedynką oznaczone są obserwacje w których obserwacja przyjmuje tą zmienną, a zerem wszystkie pozostałe.

Obraz zawierający stół

Opis wygenerowany automatycznie

Po przekształceniu otrzymujemy zbiór danych z 637 kolumnami. Z czego 2 kolumny z danymi typu float64, 5 z typem int64 i 630 uint8. Dane powinny być wstępnie przygotowane i gotowe do przekazania do wybranego modelu sieci neuronowej.

Obraz zawierający tekst

Opis wygenerowany automatycznie

## 3.2. Wybór i uczenie modelu

Pierwszym krokiem w tworzeniu modelu, jest podział danych na zbiór uczący i testowy. Na początku zrobimy to w stosunku 80/20. W późniejszych krokach będziemy sprawdzali czy zmiany tych proporcji pomogą nam uniknąć ewentualnego zjawiska niedotrenowania/przetrenowania modelu.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Kolejnym krokiem jest zadeklarowanie modelu i jego parametrów. Zdecydowaliśmy się na wybór typu modelu Sequential() dostępnego w bibliotece Keras. Działa on na zasadzie swego rodzaju schematu, do którego możemy dodawać kolejne warstwy neuronów z określonymi przez nas parametrami.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Na początku zdecydowaliśmy się na stworzenie prostego modelu, składającego się z 3 warstw. Jednej wejściowej z X z 16 punktami wyjścia i 634 punktami wejścia danych, wybrano funkcję aktywacji ‘relu’ (rectified linear activation function) która jest jedną z najczęściej wybieranych funkcji aktywacji w warstwach (poza wyjściową). Funkcja relu przekazuje dalej wynik jeżeli jest pozytywny, natomiast jeżeli jest mniejszy od zera przekaże do kolejnego neurona zero. Druga warstwa (w naszym przypadku jedyna warstwa ukryta) ma tą samą funkcję aktywacji, 32 punkty wyjściowe. Natomiast ostatnia trzecia warstwa, warstwa wyjściowa ma 3 punkty wyjścia (tyle ile zmiennych w y), ponieważ przewidujemy jeden z trzech wyników:

1. wygrał biały
2. wygrał czarny
3. remis

Funkcja softmax daje prawdopodobieństwa dla każdej z trzech możliwości, sumujące się do jedynki. Opcja z najwyższym prawdopodobieństwem oznaczana jest jako jeden a pozostałe zerami, w ten sposób otrzymujemy przypisanie obserwacji do danej klasy.

**Optymalizer**

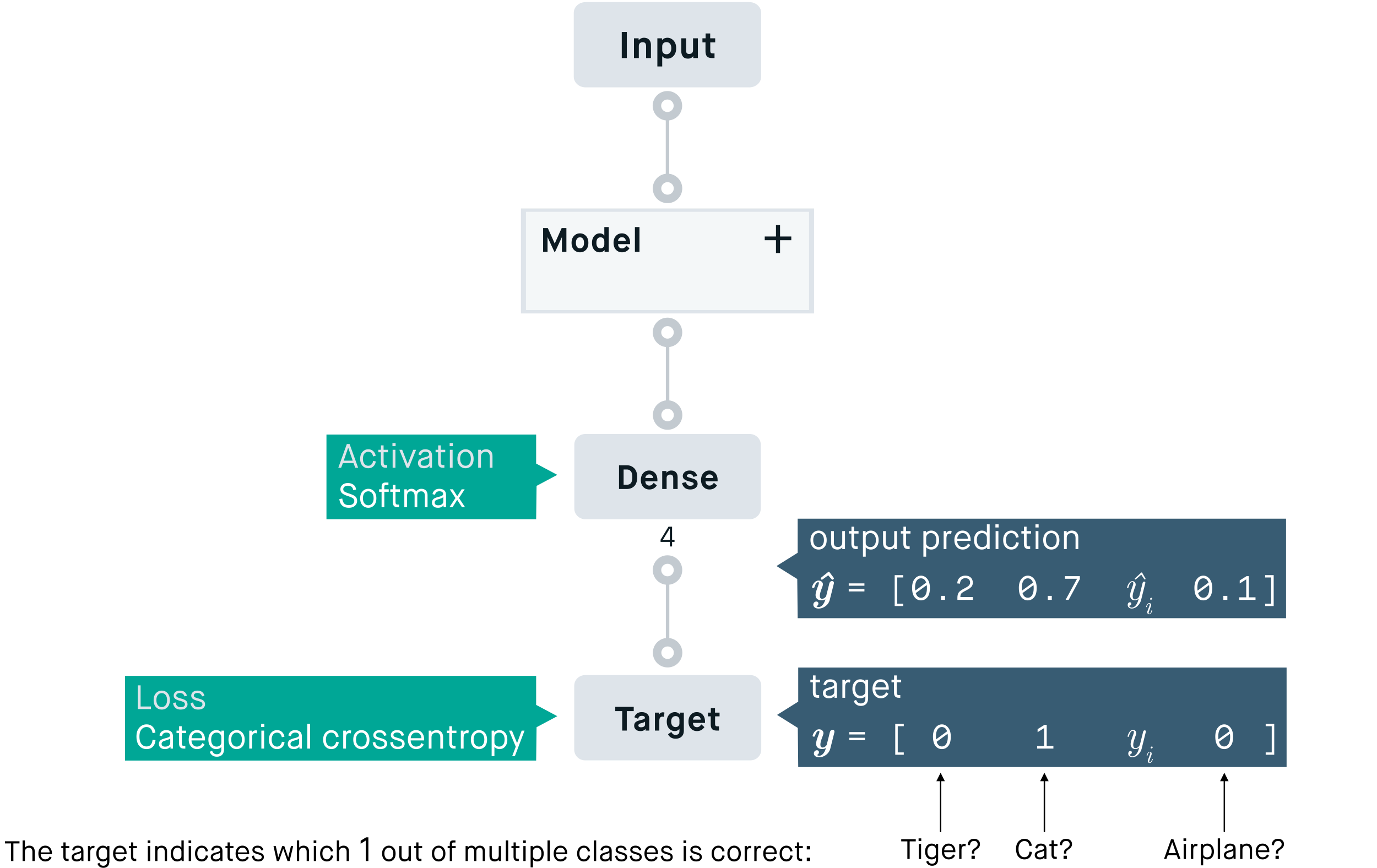
Jako optymalizer wybraliśmy najczęściej zalecany optymalizer Adam, który adaptuje uczenie się, poprzez małe zmiany dla często występujących wartości oraz duże zmiany dla rzadziej występujących. Optymalizer Adam zapisuje także poprzednie zmieny wartości, dzięki czemu w kolejnych iteracjach może lepiej dopasowywać zmiany w uczeniu.

**Funkcja straty**

Nie możemy obliczyć idealnych wag dla sieci neuronowej; jest zbyt wiele niewiadomych. Zamiast tego, problem uczenia się jest rzutowany na problem wyszukiwania lub optymalizacji, a algorytm jest używany do nawigacji w przestrzeni możliwych zestawów wag, których model może używać w celu wykonania dobrych lub dostatecznie dobrych prognoz.

Funkcja, którą chcemy zminimalizować lub zmaksymalizować, nazywana jest funkcją celu lub kryterium. Kiedy ją minimalizujemy, możemy ją również nazwać funkcją kosztu, funkcją straty lub funkcją błędu.[[7]](#footnote-7)

Jedną z funkcji straty, polecanych do użycia w przypadku problemów obejmujących klasyfikację wielu zmiennych, była funkcja categorical\_crossentropy. Funkcja ta przypisuje wartość prawdopodobieństwa 1 do tych zmiennych, które są najbardziej prawdopodobne (na podstawie wyników z wyjściowej warstwy neuronów).



Skompilowanie modelu z wybraną funkcją straty, optymajzerem i metryką która będzie zwracana (w naszym przypadku categorical\_accuracy)

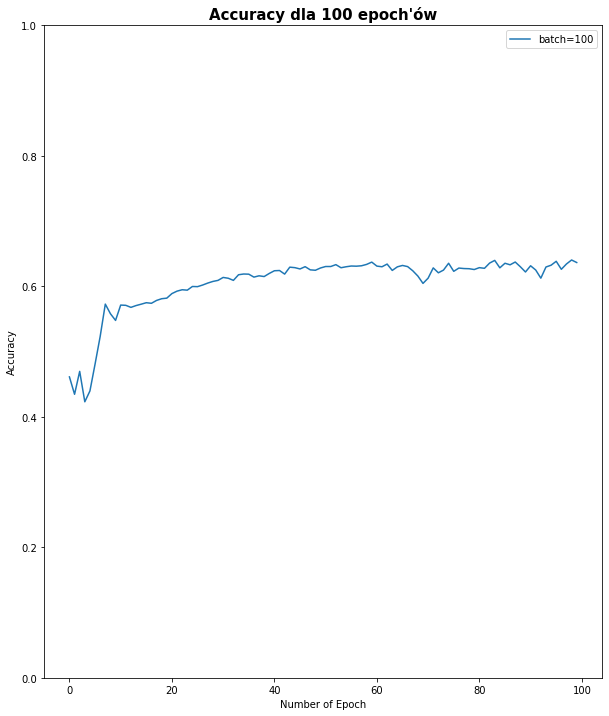


Categorical accuracy zwraca jak często predykcja się zgadzała z wartością rzeczywistą (używana często w problemach klasyfikacji, dla zmiennych zakodowanych przy użyciu one hot encoder).

Kolejnym krokiem jest fitowanie, czyli uczenie modelu przy użyciu uczącej części zbiorów. Na początku uczyliśmy model przy hiperparametrach *epcochs* i *batch\_size* ustawionych odpowiednio na 100 i 1000. Paramter *epochs* odpowiada za to ile razy podczas uczenia model przejdzie przez wszystkie wiersze, a *batch\_size* co ile wierszy dostosowywane będą wagi w modelu. W naszym przypadku oznacza to że podczas uczenia algorytm przejdzie 100 razy przez wszystkie wiersze i co 1000 wierszy będzie dostosowywał wagi w funkcjach aktywacji.



W pierwszych przejściach można się oczywiście spodziewać niższej dokładności, a wraz z każdym kolejnym epoch’iem model powinien zyskiwać na dokładności. Jest to widoczne na poniższym wykresie gdzie porównaliśmy wartości categorical\_accuracy w odniesieniu do kolejnych epochów.



Dokładność modelu dla pierwszych epochów oceniona została na wartości w okolicy 0.46, natomiast dla ostatnich 2 epochów otrzymane wartości były już bliskie 0.64

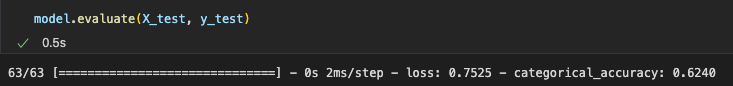
Obraz zawierający tekst

Opis wygenerowany automatycznie

Obraz zawierający tekst

Opis wygenerowany automatycznie

Następnie sprawdziliśmy wydajność modelu na zbiorze testowym:



Uzyskliśmy dokładność na poziomie 0.624, oznacza to że dla 62,4% obserwacji przydzielony został poprawny wynik. Średnia strata wyniosła 75,3%.

## 3.3. Test modelu ze zmienioną zmienną objaśniającą

Wynik dokładności uzyskany w modelu przewidującym wygraną białego, czarnego lub remis był zadowalający, ale zdecydowaliśmy się zmienić podejście w celu uzyskania wyższego wyniku. Uprościmy model do predykcji czy biały wygra, czy też nie. W pewnym sensie odrzucamy problem remisów, traktując je jako przegraną dla białego. Liczba remisów w całym zbiorze nie była duża, ale na pewno nie była zerowa, więc takie podejście powinno pomóc sieci w lepszej ocenie szansy białego na wygranie.



Tworzymy model analogiczny do poprzedniego, z trzema warstwami i identycznymi specyfikacjami warstw (oprócz wyjściowej gdzie zmieniamy rozmiar rozmiar wyjścia na 1). Jedną wejściową, jedną ukrytą i jedną wyjściową.

Obraz zawierający tekst

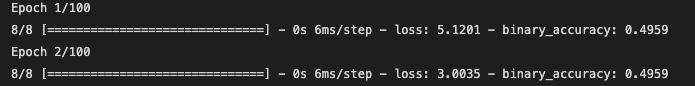
Opis wygenerowany automatycznie

Pozostajemy przy tym samym optymalizerze (Adam), zmieniamy tylko funkcję straty na binary\_crossentropy i metrykę wyniku na binary\_accuracy. Teraz nie przeprowadzamy już uogólnionej klasyfikacji, a jedynie klasyfikację binarną, więc te zmiany teoretycznie nie są konieczne, aczkolwiek możliwe że pomogą modelowi w lepszym uczeniu.

Przeprowadzamy uczenie modelu z takimi samymi hiperparametrami, *epochs=100* i *batch\_size=1000.*



Pierwsze iteracje dały accuracy na poziomie 49,6% i jest to wynik identyczny jak w ostatnich. Udało się jedynie poprawić średni wynik błędu. Wartość *loss* spadła z 5,12 do 0.626 w ostatnim epochu



Obraz zawierający tekst

Opis wygenerowany automatycznie

Wnioskujemy, że poprzedni model w którym przewidywaliśmy wygraną białego, czarnego lub remis miał wyższy poziom dokładności. W związku z tym w dalszych walidacjach pozostaniemy przy początkowym modelu.

## 3.4. Skuteczność modelu w zależności od zmiany liczby epoch i batch size

Do sprawdzenia wpływu różnych wartości hiperparametrów wykorzystamy klasę GridSearchCV, która sprawdza wynik dokładności modelu dla każdej z kombinacji przekazanych parametrów. Ze względu na to, że klasa to pochodzi z biblioteki sci-kit learn musimy zbudować funkcję która będzie generowała model.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Teraz zadeklarujemy paramatry dla których kombinacji chcemy sprawdzić wyniki modelu. Będziemy sprawdzali dla epochów ze 20, 60, 100, batch size równych 100, 500, 1000 i 2000 oraz liczby neuronów (*unit*) 8 i 16. Ograniczamy listę liczebności neuronów aby zyskać na czasie, ponieważ już te kombinacje parametrów dadzą nam 3 \* 4 \* 2 = 24 różne kombinacja. Uczenie pojedynczego modelu w naszym przypadku zajmowało ok. 15 sekund. Najpierw więc ustalimy najlepszą liczbę epcho’s i batch size a dopiero potem dla niej przeprowadzimy badanie najlepiej liczby neuronów.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Przeprowadzamy uczenie:



Najlepszą kombinacją parametrów okazało się epochs: 20, batch size: 500 i unit: 16

Obraz zawierający tekst, droga, urządzenie, tablica wyników

Opis wygenerowany automatycznie

## 3.5. Skuteczność modelu w zależności od liczby neuronów

Ponownie przy użyciu GridSearchCV przeprowadzimy badanie wpływu liczby neuronów na osiągany wynik dokładności modelu. Tym razem w modelu zastosujemy już najlepsze wybrane parametry epochs i batch size równe odpowiednio 20 i 500. Jedynie zmienna unit, czyli liczba nerunów będzie przyjmowała wartości 8, 16, 32, 64, 128, 256, 512.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Uczenie:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Po posortowaniu wyników od najwyższej dokładności do najniższej otrzymaliśmy następujące wyniki:

Obraz zawierający tekst, droga, czarny, tablica wyników

Opis wygenerowany automatycznie

Liczbą neuronów w warstwie wejściowej i ukrytej, która gwarantuje najwyższy wynik jest 16.

Dała wynik accuracy na poziomie 62,5%, podobnie jak kolejna opcja 32 neuronów. Natomiast 3 opcja 64 neuronów zanotowała już spadek dokładności modelu do 60,3%.

Dlatego też w dalszym testowaniu modelu pozostaniemy przy 16 neuronach w każdej z warstw (oprócz wyjściowej)

## 3.6. Skuteczność modelu w zależności od liczby warstw ukrytych

Budujemy model z najlepszymi wyznaczonymi parametrami, czyli:

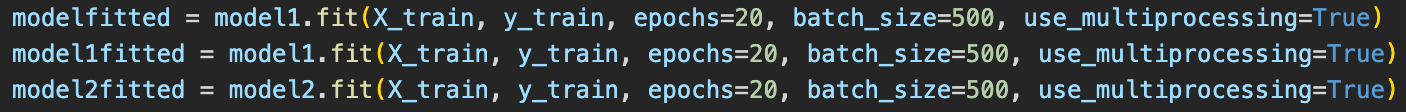
* epochs = 20
* batch\_size = 500
* unit = 16

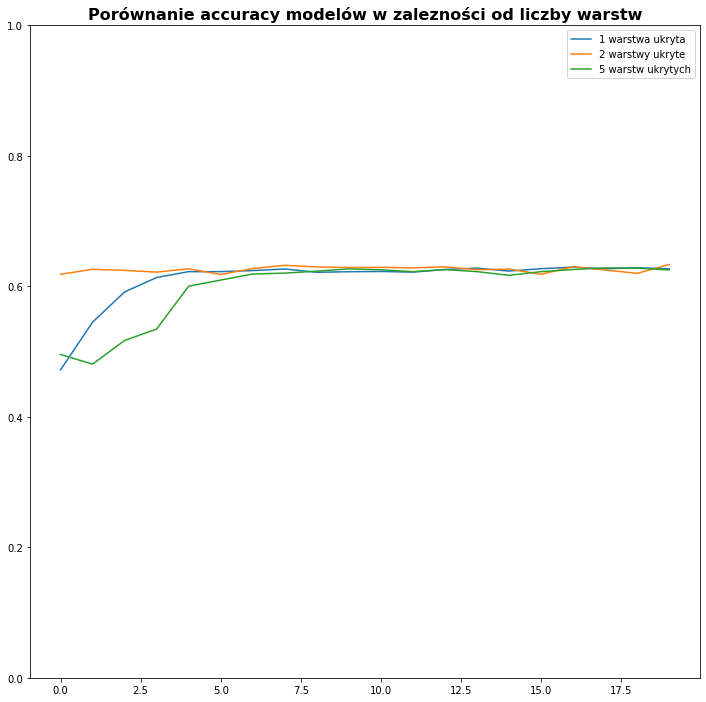
ale z różnymi liczbami warstw ukrytych. Pierwszy z nich będzie miał 2 warstwy ukryte a drugi 5 warstw ukrytych. Porównamy dokładności tych modeli z modelem o jednej warstwie ukrytej.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Uczenie modeli:



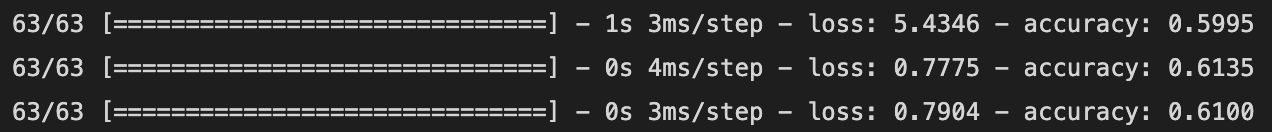


Można zauważyć, że model z 2 warstwami ukrytymi poradził sobie najlepiej przy pierwszych iteracjach epoch, natomiast od 7/8 iteracji wszystkie modele wykazywały praktycznie taką samą dokładność. Nie udało się uzyskać znacząco lepszej wyników poprzez dodanie większej ilości warstw neuronowych.

Ostateczny wynik modelu na zbiorze testowym:

Obraz zawierający tekst

Opis wygenerowany automatycznie



Najlepszy wynik na zbiorze testowym osiągnął model z 2 warstwami z wynikiem 61,3% dokładności. Model z 5 warstwami ukrytymi uzyskał 61% a początkowy z jedną warstwą ukrytą 60%.

1. *Jak są zbudowane sieci neuronowe?* <https://course.elementsofai.com/pl/5/2> [↑](#footnote-ref-1)
2. *Czym jest deep learning i sieci neuronowe* <https://bulldogjob.pl/articles/1136-czym-jest-deep-learning-i-sieci-neuronowe?fbclid=IwAR184dhH63WkxhwPHCJZcnVjEIrricTQzd0aJ_IGbFwzxYDzgC0LXRqN1SU> [↑](#footnote-ref-2)
3. *Algorytmy uczenia sieci neuronowych* <http://www.neurosoft.edu.pl/media/pdf/tkwater/sztuczna_inteligencja/2_alg_ucz_ssn.pdf> [↑](#footnote-ref-3)
4. *„End-to-End Deep Neural Network for Automatic Learning in Chess”* <https://www.researchgate.net/profile/Konstantin-Herud/publication/330317757_End-to-End_Deep_Neural_Network_for_Automatic_Learning_in_Chess/links/5fbacb82299bf104cf6ce5bd/End-to-End-Deep-Neural-Network-for-Automatic-Learning-in-Chess.pdf> [↑](#footnote-ref-4)
5. *„Predicting Moves in Chess using Convolutional Neural Networks”* <http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf> [↑](#footnote-ref-5)
6. *Encoding cyclical continuous features - 24-hour time*, <https://ianlondon.github.io/blog/encoding-cyclical-features-24hour-time/>, (25.05.2022) [↑](#footnote-ref-6)
7. <https://www.brutalk.com/pl/wiadomosci/brutalk-blog/przeglad/funkcje-strat-i-strat-do-szkolenia-sieci-neuronowych-uczenia-glebokiego-6046fa42120e8> , (25.05.2022) [↑](#footnote-ref-7)