Sprawozdanie z laboratorium:
Uczenie maszynowe

Uczenie nadzorowane warstwowych sieci neuronowych

21 maja 2020

Prowadzący: dr hab. inż. Maciej Komosiński

Autorzy:	**Bartosz Mikulski**	inf131803	ISWD	bartosz.j.mikulski@student.put.poznan.pl
	Szymon Janowski	inf131770	ISWD	szymon.p.janowski@student.put.poznan.pl

Zajęcia odbywające się w środę o godzinie 15:10.

Oświadczam/y, że niniejsze sprawozdanie zostało przygotowane wyłącznie przez powyższych autora/ów, a wszystkie elementy pochodzące z innych źródeł zostały odpowiednio zaznaczone i są cytowane w bibliografii.

Udział autorów

- Szymon Janowski – przeprowadził eksperymenty oraz opisał zadania 3, 4, 5 z części pierwszej laboratoriów oraz zadania 9 i 10 z drugiej części laboratoriów (w sprawozdaniu podpunkty: 1, 2, 3, 6, 7).
- Bartosz Mikulski – przeprowadził eksperymenty oraz opisał 7, 8, 10, 11, 12 z drugiej części laboratoriów (w sprawozdaniu podpunkty 4, 5, 7, 8, 9).
- Wszyscy autorzy przeczytali i zaakceptowali kompletne, ostateczne sprawozdanie.

1 Różnice funkcjonalne pomiędzy sieciami jedno- i wielowarstwowymi oraz pomiędzy sieciami liniowymi a nieliniowymi.

1.1 Zapoznanie się z narzędziami udostępnionymi do realizacji laboratoriów.

Jako materiały pomocnicze, udostępniony został notatnik programu *Jupyter* [5] oraz skrypt języka *Python* [6] *NN_helpers* zawierający funkcje implementujące prezentację wytworzonej sieci neuronowej z wszystkimi wyliczonymi wagami jak i grafikę prezentującą predykcje sieci dla podanego przykładu zarówno na schemacie 2D jak i 3D. Notatnik *NN_cwiczenia* zawiera przykładowe implementacje modelu *MLP (Multi Layer Perceptron)* [7] sieci neuronowej, dla różnych zbiorów danych i parametrów uczenia. Wykorzystywany klasyfikator zaimplementowany został w bibliotece *Scikit–Learn* [10]. Notatnik posiada też prezentacje możliwości funkcji zdefiniowanych w poprzednio wspomnianym skrypcie języka *Python NN_helpers*.

1.2 Konstrukcja zbioru przykładów definiujący dwuargumentową funkcję ("bramkę") *AND*.

Dane treningowe składają się z 4 przykładów uczących posiadających po dwa atrybuty. Każdy przykład przyporządkować można do jednej z dwóch klas decyzyjnych (0 lub 1). Trzy przykłady można przyporządkować do etykiety 0, a jedną do etykiety 1.

Zbiór przykładów uczących: X = [[0,0],[0,1],[1,0],[1,1]]
Zbiór etykiet dla przykładów uczących: y = [0,0,0,1]

Na podstawie zbioru uczącego narysować można pożądaną funkcję odpowiedzi sieci dla wszystkich wartości (Rys. 1).

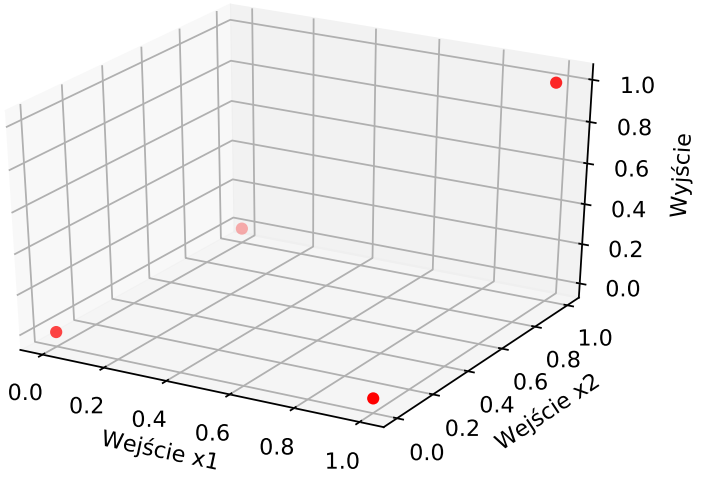
1.3 Konstrukcja liniowej sieci jednowarstwowej dla problemu *AND*.

Wymagana do utworzenia sieć neuronowa o architekturze 2–1 składa się z warstwy ukrytej z dwoma neuronami z funkcją aktywacji *identity* oraz warstwą wyjściową z jednym neuronem z funkcją aktywacji *logistic* [7]. Sieć nauczona została na danych treningowych z Podrozdziału 1.2. Z użyciem funkcji *warm start* [7] podczas procesu uczenia zebrane zostały dane pozwalające na narysowanie funkcji straty (ang. *loss*) i odpowiedzi sieci dla poszczególnych przypadków (Rys. 2 i 3). Naturalne jest to, że przy zwiększaniu liczby iteracji funkcja straty maleje, czyli wagi neuronów są coraz bardziej pewne. Jednocześnie Rys. 3 pokazuje, że wszystkie przypadki uczące znajdują się w obszarach gdzie zostały poprawnie zaklasyfikowane. Czyli na zbiorze treningowym sieć posiada 100% skuteczności.

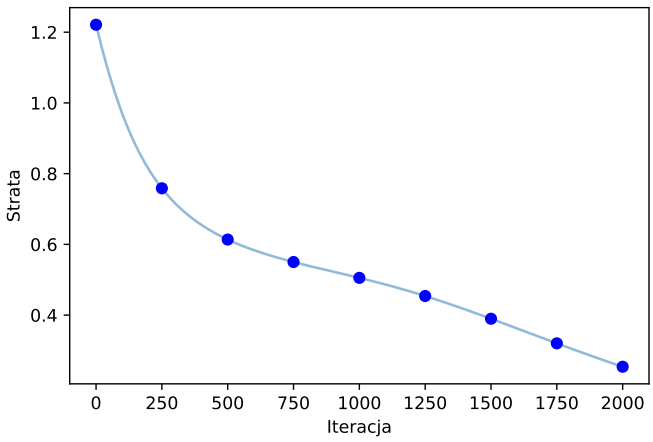
Zmiana architektury sieci, gdzie warstwa ukryta posiadać będzie 5 (zamiast 2) neuronów, nie zmieniła skuteczności sieci (nadal jest ona 100%). Natomiast, jak pokazuje Rys. 4, sieć jest pewniejsza (czyli prawdopodobieństwo wyboru danej klasy wzrośnie) co do predykcji dla przykładów ze zbioru uczącego.

1.4 Konstrukcja nieliniowej sieci jednowarstwowej dla problemu *AND*.

Zmiana architektury z Podrozdziału 1.3 tak, żeby reprezentowała konstrukcję nieliniową wymaga jedynie zmiany funkcji aktywacji neuronów warstwy ukrytej z *identity* na np. *logistic*.

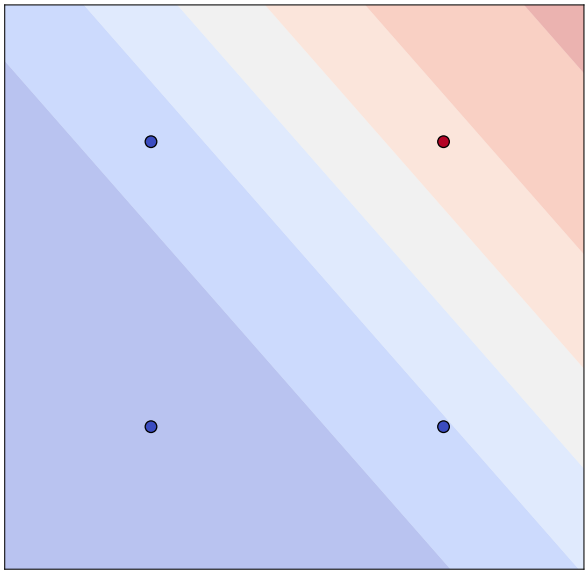


Rysunek 1: Pożądana funkcja odpowiedzi sieci dla 4 wartości ze zbioru uczącego problemu *AND*. Jest to wykres zależności wyjścia sieci neuronowej od dwóch wejść.



Rysunek 2: Funkcja straty w czasie 2000 iteracji, w których uczona była sieć.

Tak przygotowana sieć nauczona została na problemie *AND*. Zachowanie sieci nie uległo zmianie, także poprawność jej predykcji się nie zmieniła. Konstrukcja nieliniowa, nie wpływa znacznie na ten problem, ze względu na to, że konstrukcja nieliniowa akurat w tym przypadku, jest w stanie dobrze opisać problem *AND*, co jednak nie zdarza się dla każdego zbioru danych. Dlatego też liniowe funkcje aktywacji są rzadko używane.



Rysunek 3: Funkcja odpowiedzi sieci dla przypadków uczących. Sieć posiada architekturę 2–1. Punkty oznaczone kolorem niebieskim są przykładami posiadającymi etykietę 0, natomiast czerwonym zaznaczony jest przypadek posiadający etykietę 1. Kolory widoczne na rysunku mówią o prawdopodobieństwie przynależności punktu do danej klasy gdy (poprzez jego atrybuty) znajdowałby się w tym obszarze (kolory mają znaczenie analogiczne do wcześniej wspomnianych punktów).

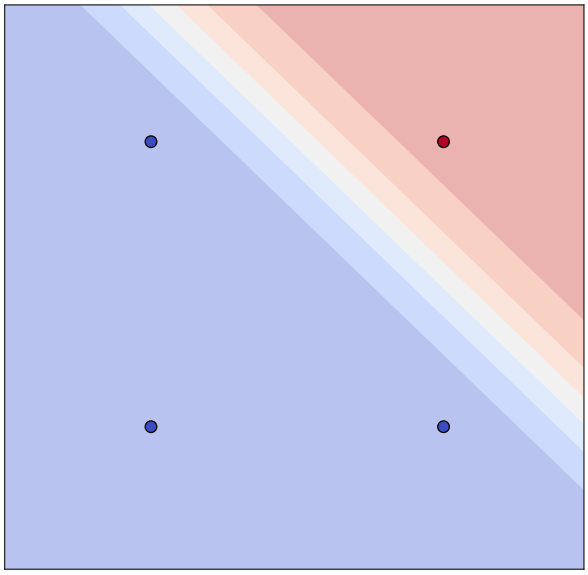
1.5 Wyznaczenie w przestrzeni wejść prostą, która definiuje neuron wyjściowy. Ocena separacji klas decyzyjnych przez tą funkcję.

Na Rys. 5 przedstawiona jest podzielona przestrzeń wejść. Funkcja dzieląca tą przestrzeń definiowana jest na podstawie wag (razem z obciążeniem (ang. *bias*)) neuronu wyjściowego. Funkcja liniowa oddziela przykład wejściowy $[0, 0]$, któremu model przyporządkował etykietę 0. Jest to przykład z największym prawdopodobieństwem wystąpienia tej etykiety. Funkcja ta separuje przykłady $[0, 1]$ i $[1, 0]$ od wcześniej wspomnianej $[0, 0]$.

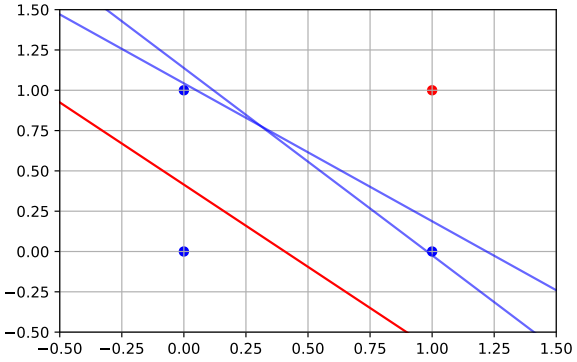
1.6 Konstrukcja zbioru przykładów definiujący dwuargumentową funkcję ***XOR***.

Dane treningowe posiadają 4 przykłady uczące posiadające po dwa atrybuty. Każdy przykład przyporządkować można do jednej z dwóch klas decyzyjnych (0 lub 1). Dwa przykłady można przyporządkować do etykiety 0, dwie pozostałe do etykiety 1.

Zbiór przykładów uczących: X = [[0,0],[0,1],[1,0],[1,1]]
Zbiór etykiet dla próbek uczących: y = [0,1,1,0]



Rysunek 4: Funkcja odpowiedzi sieci dla przypadków uczących. Sieć posiada architekturę 5–1.

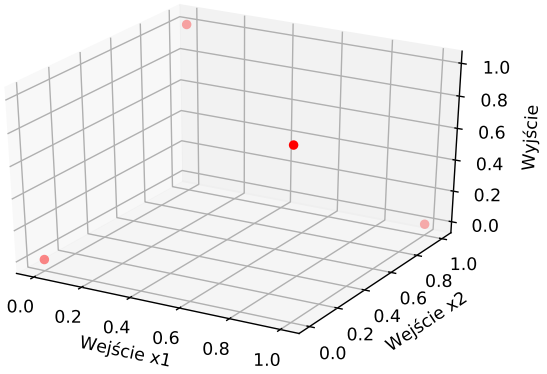


Rysunek 5: Prosta wyznaczona w przestrzeni wejść. Definiuje ona neurony architektury sieci rozwiązującej problem *AND*. Kolorem niebieskim zaznaczone są funkcje określone na podstawie wag neuronów warstw ukrytych. Kolorem czerwonym zaznaczona jest funkcja określona na podstawie wag neuronu wyjściowego.

Na podstawie zbioru uczącego narysować można pożądaną funkcję odpowiedzi sieci dla wszystkich wartości (Rys. 6).

1.7 Konstrukcja liniowej sieci jednowarstwowej dla problemu *XOR*.

Dla problemu *XOR* skonstruowana została siec o architekturze 2–1. Wykonane zostało na niej 2000 iteracji, tak jak dla poprzednio rozpatrywanego problemu *AND*. Funkcja odpowiedzi sieci dla przypadków uczących, przedstawiona na Rys. 7, znacząco się zmieniła. Wynika z niej, że sieć błędnie interpretuje 2 przypadki czyli połowę zbioru uczącego. Potwierdza to także wywołanie funkcji *predict* [7] utworzonego modelu dla przykładów treningowych.



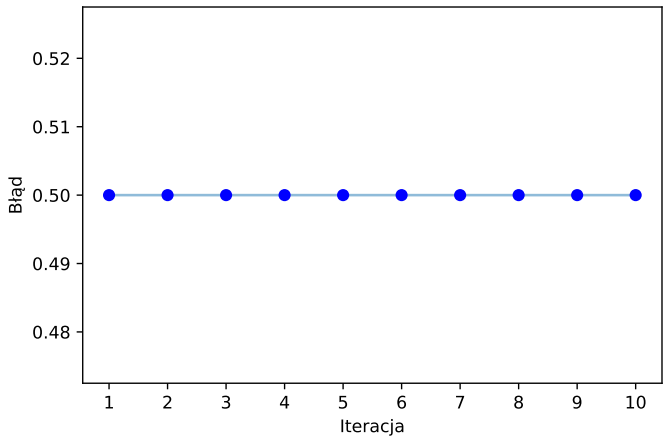
Rysunek 6: Pożądana funkcja odpowiedzi sieci dla 4 wartości ze zbioru uczącego problemu *XOR*. Jest to wykres zależności wyjścia sieci neuronowej od dwóch wejść.

1.8 Konstrukcja nieliniowej sieci jednowarstwowej dla problemu *XOR*.

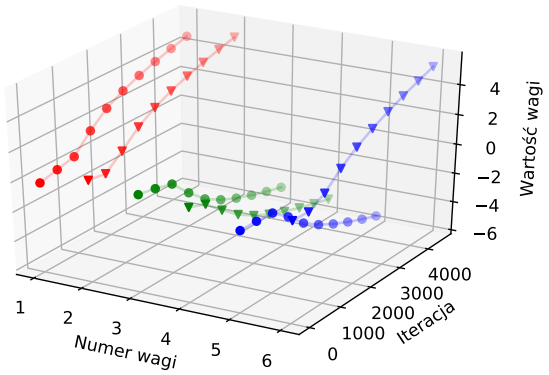
Po przerobieniu jednowarstwowej sieci zdefiniowanej dla problemu *XOR* na nieliniową, sieć nauczona została na tych samych przykładach treningowych. Liczony też był błąd w czasie 10 iteracji uczenia modelu. Rys. 7 przedstawia zmianę błędu, który jest liczony jako $1 - a$, gdzie a oznacza trafność predykcji na zbiorze treningowym. Taka liczba iteracji jest zbyt mała by błąd się zmienił i pozostaje on taki sam jak w konstrukcji liniowej. Dodatkowo zwiększenie liczby iteracji może zmniejszyć błąd. Problem *XOR* nie może być reprezentowany przez pojedynczy perceptron, jednak w tej architekturze są 2 neurony w warstwie ukrytej i jeden neuron wyjściowy (architektura 2–1) dlatego też przy odpowiednim dostosowaniu parametrów, możliwe jest uzyskanie 100% skuteczności dla problemu *XOR*. W tym eksperymencie badane też zostały wagi neuronów podczas kolejnych iteracji i przedstawione w formie Rys. 8. Na Rys. 8 przedstawione jest wszystkie 6 wag sieci (po dwie dla każdego neuronu z warstwy ukrytej). Wagi jednego neuronu oznaczone są tym samym kolorem.

1.9 Konstrukcja nieliniowej sieci dwuwarstwowej dla problemu *XOR*.

Architektura sieci realizującej problem *XOR* zmieniona została na następujący podział warstw ukrytych (łącznie z wyjściową): 2–2–1. Jako parametr dotyczący funkcji aktywacji ustawiony

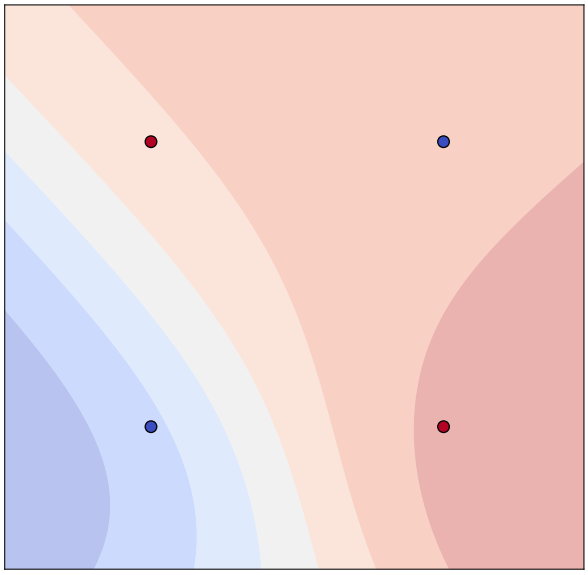


Rysunek 7: Funkcja błędu dla 10 iteracji sieci dla problemu *XOR*.



Rysunek 8: Zmiana wartości poszczególnych wag w 10 kolejnych etapach uczenia, podczas 10000 iteracji. Wagi dla tych samych neuronów oznaczone są tym samym kolorem.

został *logistic*. Podczas procesu uczenia wykonane zostało 2000 iteracji. Eksperyment ten nie dał jednak zadowalającego rezultatu (Rys. 9). Zmiany wymagały parametry *solver* i *learning_rate_init*, które kolejno odpowiadają za optymalizator wag i krok aktualizacji wag. Parametr *solver* ustawiony został na optymalizator *adam* [1], natomiast *learning_rate_init* ustawiony został na wartość 0.1. W ten sposób sieci udało się zrealizować problem *XOR* bez błędów klasyfikacji co przedstawia Rys. 10. Modyfikując ziarno losowe przy inicjalizacji klasyfikatora, przy tych samych parametrach można uzyskać inne wyniki funkcji reprezentującej problem *XOR* (Rys. 11).



Rysunek 9: Pożądana funkcja odpowiedzi sieci wartości ze zbioru uczącego problemu *XOR*. Jest to wykres zależności wyjścia sieci neuronowej od dwóch wejść. Jeden przypadek klasyfikowany jest niepoprawnie.

1.10 Analiza wag sieci. Interpretowanie działania neuronu wyjściowego.

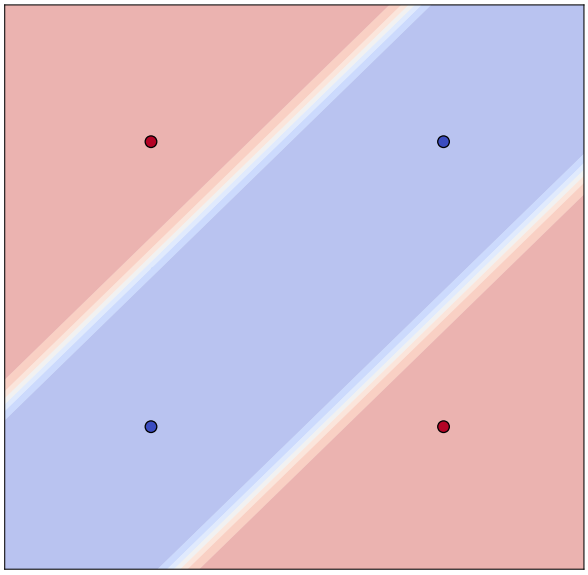
Wagi razem z obciążeniami (ang. *biases*) poszczególnych neuronów definiują proste wyznaczone w przestrzeni wejść (Rys. 12). Neuron wyjściowy działa jako ostateczny filtr dla sygnałów w sieci. Za pomocą wag wybiera najbardziej znaczące sygnały i zwraca zakodowaną wartość opisującą dane podane na wejściu sieci. Wartość taka może być interpretowana jako prawdopodobieństwo przynależności do klasy 1.

Wagi sieci zmieniały się cały czas podczas procesu uczenia. Zmianę wartości poszczególnych wag w kolejnych iteracjach obrazuje Rys. 13. Wagi tego samego neuronu zaznaczone są tym samym kolorem.

2 Obserwacja zjawiska przeuczenia na przykładzie zbioru *PIMA*.

2.1 Zapoznanie się z pochodzeniem zbiorów *PIMA* i *BUPA*.

Zbiór danych *PIMA* dotyczy historii chorób plemienia Indian Pima, a dokładniej cukrzycy. Posiada 8 atrybutów i jedną kolumnę dotyczącą binarnej klasy atrybutów (chory na cukrzycę – 1 lub nie – 0). Ilość próbek wynosi 767.



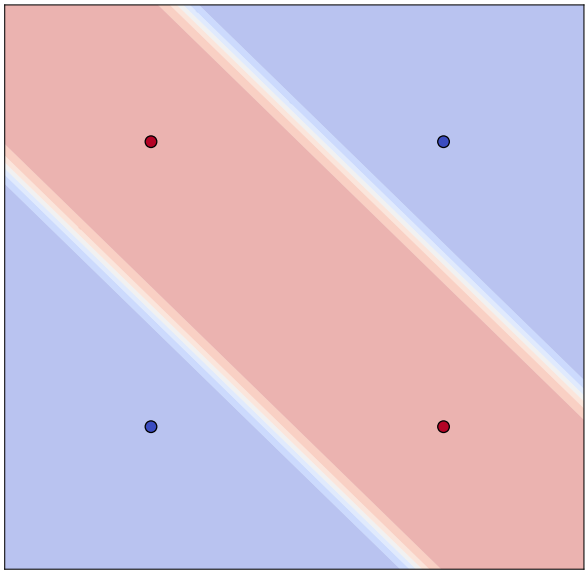
Rysunek 10: Pożądana funkcja odpowiedzi sieci wartości ze zbioru uczącego problemu *XOR*. Jest to wykres zależności wyjścia sieci neuronowej od dwóch wejść. Wszystkie przypadki klasyfikowane są poprawnie.

Zbiór danych *BUPA* dotyczy zaburzeń wątroby. Posiada 7 atrybutów dotyczących między innymi liczby spożywanych napojów alkoholowych w ciągu dnia czy średniej objętości krwinek (ang. *mcv*). Ilość próbek wynosi 345.

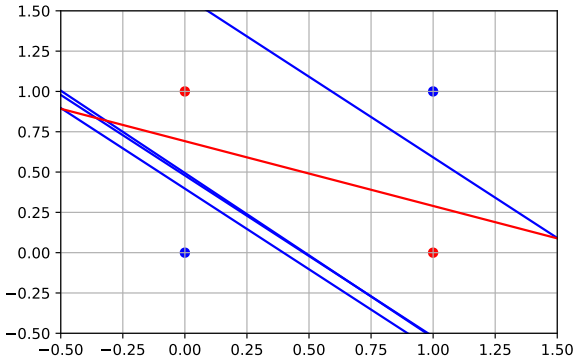
2.2 Wczytanie zbioru *PIMA* i skonstruowanie dla niego dwuwarstwową sieć nieliniową.

Dane wczytane zostały z pliku o rozszerzeniu *CSV* za pomocą biblioteki *Pandas*. Nazwy kolumn zostały podane jako dodatkowy parametr podczas wczytywania zbioru danych. Dane podzielone zostały na zbiór przypadków uczących opisywanych przez atrybuty problemu i osobny zbiór etykiet dla tych przypadków. Etykiety znajdowały się w kolumnie *class*. Zbiór danych *PIMA* podzielony został na część treningową (90% wszystkich przypadków) i testową (10% wszystkich przypadków) za pomocą funkcji *train_test_split* zaimplementowanej w środowisku *Scikit–Learn*. Następnie wykonana została operacja standaryzacji [8], na podstawie danych tylko ze zbioru treningowego. Znormalizowany zbiór treningowy wykorzystany został do uczenia modelu. Zbiór testowy używany jest jedynie do ewaluacji predykcji tego modelu.

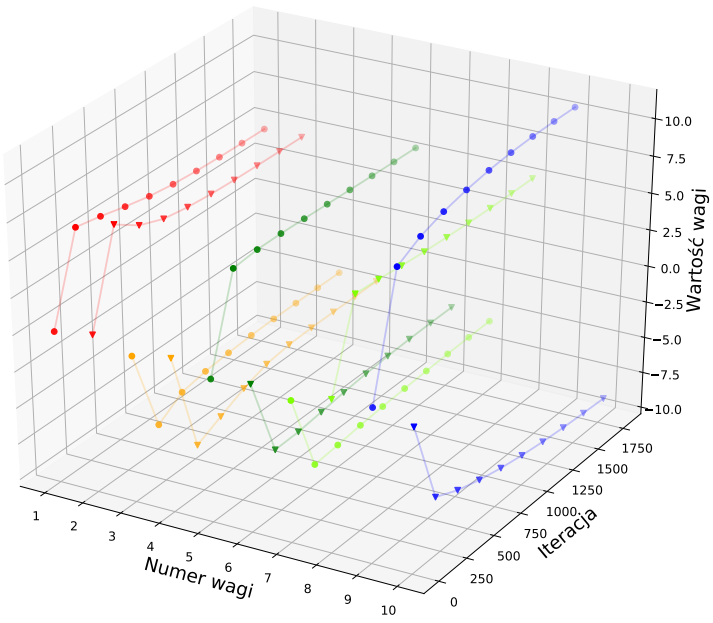
Sieć zbudowana została w architekturze 8-4-1 (Rys. 14), z nieliniową funkcją aktywacji (parametr *logistic*). Wyłączone zostały wszystkie inkrementalne warunki stopu, np. dla parametru *early_stopping* ustawiona została wartość *False*. Podczas trenowania sieci liczony jest błąd na zbiorze treningowym i testowym. Na Rys. 15 pokazany jest ten błąd zmieniający się



Rysunek 11: Alternatywny wygląd pożądanej funkcji odpowiedzi sieci wartości ze zbioru uczącego problemu *XOR*.

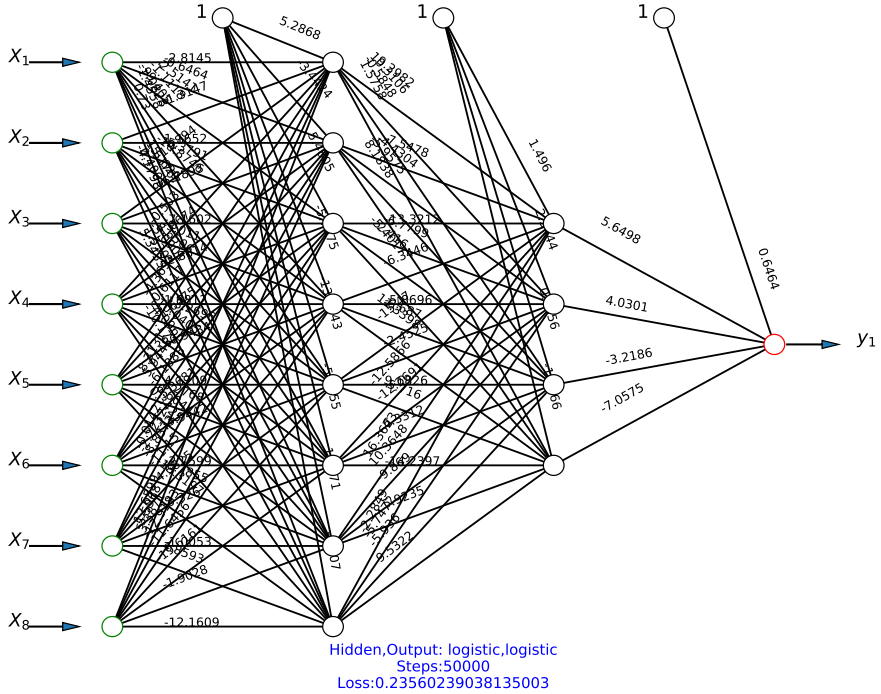


Rysunek 12: Prosta wyznaczona w przestrzeni wejść. Definiuje ona neurony architektury sieci rozwiązującej problem *XOR*. Kolorem niebieskim zaznaczone są funkcje określone na podstawie wag neuronów warstw ukrytych. Kolorem czerwonym zaznaczona jest funkcja określona na podstawie wag neuronu wyjściowego.



Rysunek 13: Zmiana wartości poszczególnych wag w 10 kolejnych etapach uczenia, podczas 2000 iteracji. Wagi dla tych samych neuronów oznaczone są tym samym kolorem. Wagi neuronów w tej samej warstwie są oznaczone zbliżonym kolorem np. czerwony – pomarańczowy.

podczas 50000 iteracji algorytmu. Zauważyć można, że do pewnego momentu błąd na zbiorze testującym maleje, a następnie zaczyna stopniowo rosnąć wraz z odpowiednio malejącym błędem na zbiorze treningowym. Można więc założyć, że model ciągle próbując poprawić swoje wyniki, ucząc się przykładów ze zbioru treningowego i tym samym niwelując tam błąd, przeucza się i traci umiejętność generalizacji. W takiej sytuacji rośnie błąd na zbiorze testowym co jest bardzo niepożądane.

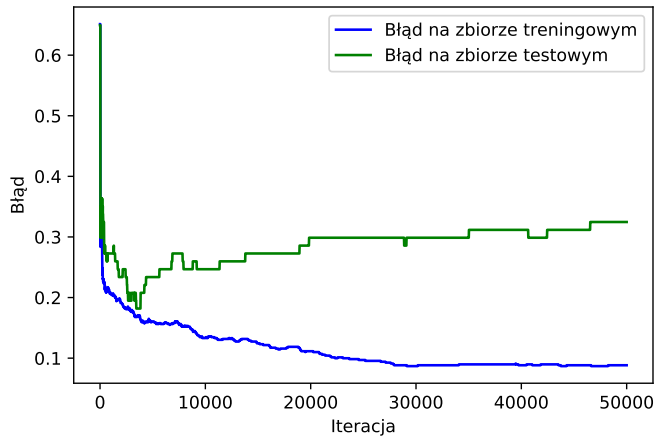


Rysunek 14: Sieć w architekturze 8-4-1 dla problemu *PIMA*.

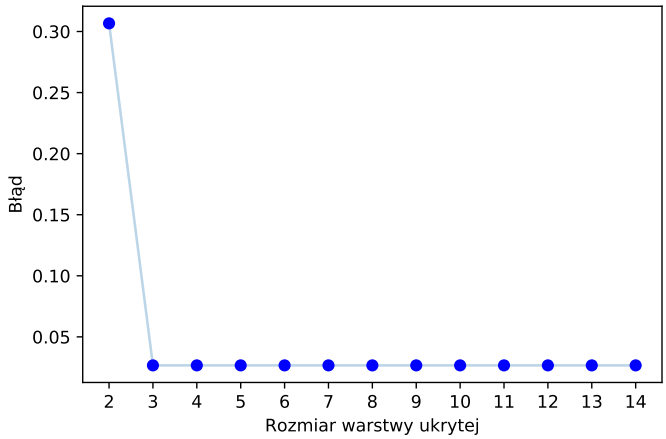
3 Dobór liczby neuronów w warstwie ukrytej na przykładzie zbioru *IRIS*.

3.1 Dla zbioru *IRIS* przeprowadzenie eksperymentów uczenia nieliniowych sieci dwuwarstwowych o architekturach 4–n–3, dla n zmieniającego się w przedziale $[2, 14]$, przy stałej założonej liczbie epok uczenia.

Zbiór danych *IRIS* został wczytany i znormalizowany. Błąd był liczony na zbiorze treningowym dla każdej sieci, po procesie uczenia trwającym zawsze 5000 iteracji bez możliwości wcześniejszego zatrzymania. Rys. 16 przedstawia wartości błędów na zbiorze treningowym

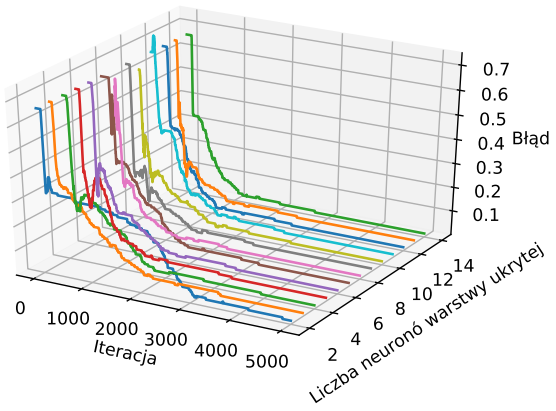


Rysunek 15: Wykres przedstawiający wartości błędu na zbiorze treningowym i testowym podczas 50000 iteracji algorytmu.



Rysunek 16: Wykres przedstawiający wartości błędów na zbiorze treningowym dla kolejnych rozmiarów warstwy ukrytej architektury sieci 4–n, gdzie n oznacza modyfikowaną liczbę neuronów w tej warstwie.

przy zastosowaniu modyfikowanego rozmiaru warstwy ukrytej zdefiniowanej w zadaniu. Wykres przedstawiony na Rys. 16 pokazuje, że dla tego przypadku i dla tego konkretnego ziarna losowego (w eksperymencie równego 42) istnieje taki rozmiar warstwy, który pozwoli na zminimalizowanie błędu na zbiorze treningowym, a nawet istnieje wiele takich wartości. Mimo to decyzja o wielkości warstwy ukrytej jest bardzo uzależniona od rozwiązywanego problemu oraz wybranych parametrów klasyfikatora. Należy też wspomnieć, że wybór optymalnej wielkości warstwy ukrytej dla tego konkretnego przypadku, weryfikowanej błędem na zbiorze treningowym nie daje wcale gwarancji najniższego błędu tego modelu na zbiorze testowym.



Rysunek 17: Wykres przedstawiający wartości błędów na zbiorze treningowym dla kolejnych rozmiarów warstwy ukrytej architektury sieci 4–n (gdzie n oznacza modyfikowaną liczbę neuronów w tej warstwie) w kolejnych iteracjach procesu uczenia.

Sieć \Rzeczywiste	0	1
0	33	17
1	9	18

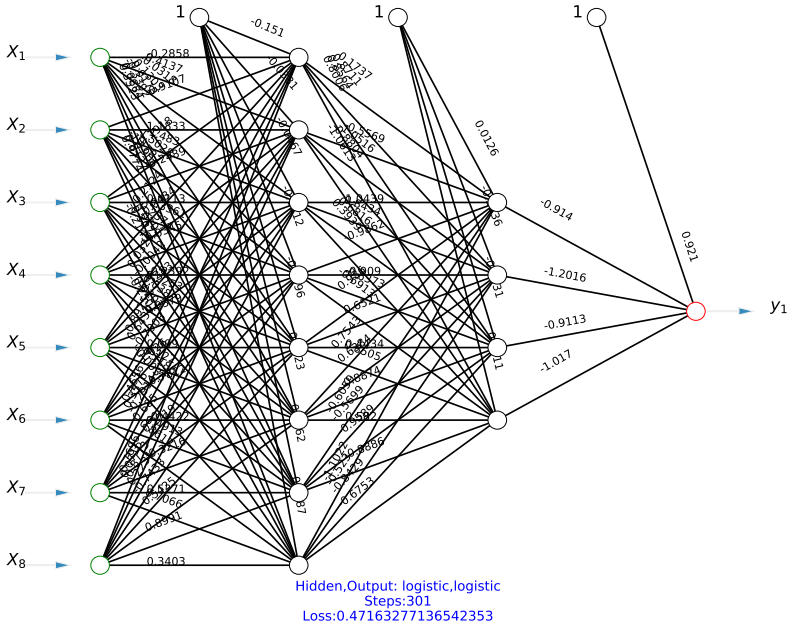
Tabela 1: Macierz pomyłek dla sieci 8–4–1 (Rys. 18) na zbiorze danych *PIMA*. Trafność klasyfikacji dla tego zbioru wynosi 66%.

Przy zwiększaniu ilości neuronów w warstwie model zyskuje większe możliwości opisu problemu (jednocześnie zwiększa się liczba wykonywanych obliczeń i proces uczenia jest wolniejszy), ale też powoduje to przeuczanie się i traci on umiejętność generalizowania. Przebieg błędów w procesie uczenia dla kolejnych architektur w tym problemie pokazany jest na Rys. 17. Dlatego bazując na Rys. 16 prawdopodobnie najlepszym wyborem byłaby warstwa ukryta o wielkości 3.

4 Sterowanie rozmiarem obszaru niepewnych odpowiedzi sieci na granicach klas decyzyjnych.

Problem dotyczy zbioru danych *PIMA*. Dane te zawierają dwie klasy decyzyjne (występuje cukrzyca lub nie występuje), a także 8 atrybutów. Wszystkie atrybuty są numeryczne. Atrybuty różnią się jeśli chodzi o interpretację, każdy z atrybutów jest w innych jednostkach. Przykłady jednostek to: wiek w latach, indeks *BMI* i grubość skóry w milimetrach.

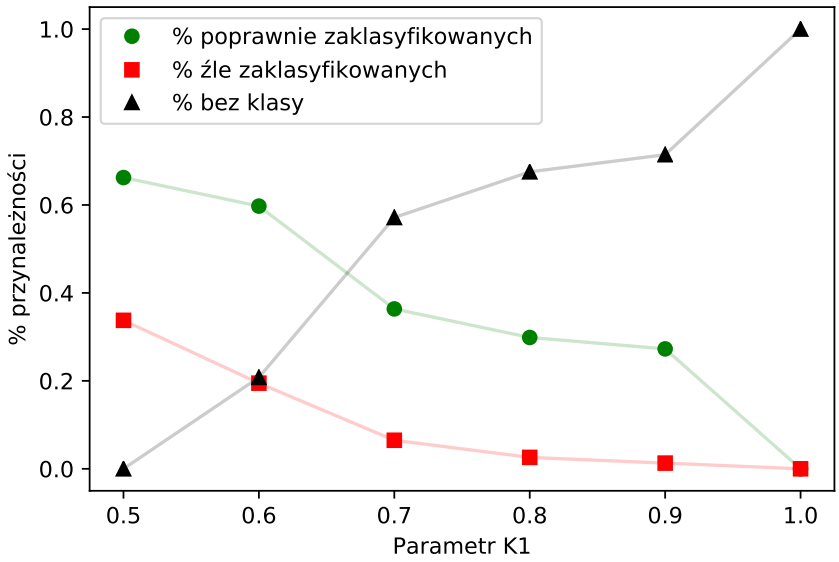
Dane zostały podzielone na zbiory treningowe i testowe, gdzie zbiór testowy zawierał 10% przypadków z oryginalnego zbioru danych. Następnie dane treningowe zostały przeskalowane za pomocą modułu *StandardScaler* [8]. Uzyskany moduł posłużył do przeskalowania danych treningowych, w taki sposób, żeby uniknąć wycieku danych. Na podstawie zbioru treningowego wytrenowano sieć neuronową w architekturze 8–4–1 widoczną na Rys. 18. Dla sieci otrzymano macierz pomyłek, patrz Tabela 1.



Rysunek 18: Sieć w architekturze 8–4–1 dla problemu *PIMA*. Sieć dla problemu *PIMA* została nauczona używając następujących parametrów: *activation*=*logistic*, *random_state*=42, *alpha*=10^{-5}.

Następnie wykonano predykcje z użyciem metody *predict_proba* [7] dla danych testowych. Metoda zwraca prawdopodobieństwo przynależności do każdej klasy. W naszym przypadku użyteczne jest prawdopodobieństwo p przynależności do klasy pozytywnej. Wtedy można zastosować zdefiniowane progi $K1$ oraz $K0$, gdzie $p \geqslant K1$ oznacza klasę pozytywną (czyli 1), a $p \leqslant K0$ oznacza klasę negatywną (czyli 0). W przypadku kiedy żaden warunek nie jest spełniony to przydzielamy sztuczną klasę „niesklasyfikowano".

Rys. 19 przedstawia procentowe przydziały do zbiorów: „dobrze zaklasyfikowanych", „źle zaklasyfikowanych" oraz „nie zaklasyfikowanych". Dla wartości $K1 = 0.5$ wykres przedstawia wartości odpowiadające macierzy pomyłek dla tego zbioru (Tabela 1). Parametr $K1$ oznacza minimalną pewność co do podjętej decyzji klasyfikatora. Dlatego dla $K1 = 0.7$, aby przypadek był zaklasyfikowany pozytywnie, to musi osiągnąć co najmniej 0.7 na wyjściu sieci neuronowej (lub co najwyżej 0.3 dla klasy negatywnej). Powoduje to, że już przy $K1 = 0.7$, więcej niż połowa przypadków nie jest w stanie być zaklasyfikowana. Oznacza to, że dla ponad połowy przypadków, sieć nie była pewna o więcej niż 70%, co do trafności osądu. Na wykresie widać



Rysunek 19: Procent przynależności do poszczególnych zbiorów w funkcji parametru $K1$. Zielone punkty oznaczają procent poprawnie zaklasyfikowanych przypadków z użyciem *predict_proba* i progów $K1$ oraz $K0$. Czerwone kwadraty odpowiadają procentowi źle zaklasyfikowanych przypadków, a czarne trójkąty oznaczają procent przykładów, które nie zostały zaklasyfikowane do żadnej z powyższych klas. Linie pomiędzy punktami są uwzględnione tylko jak pomoc dla czytelnika, pomiary były wykonywane tylko w zaznaczonych punktach.

także, że różnica pomiędzy procentem źle zaklasyfikowanych oraz dobrze zaklasyfikowanych przypadków jest w miarę stabilna dla rosnącego $K1$. Może być to spowodowane tym, że sieć dobrze klasyfikuje i myli się w podobnym stopniu dla wartości odfiltrowywanych przez parametr $K1$ oraz $K0$.

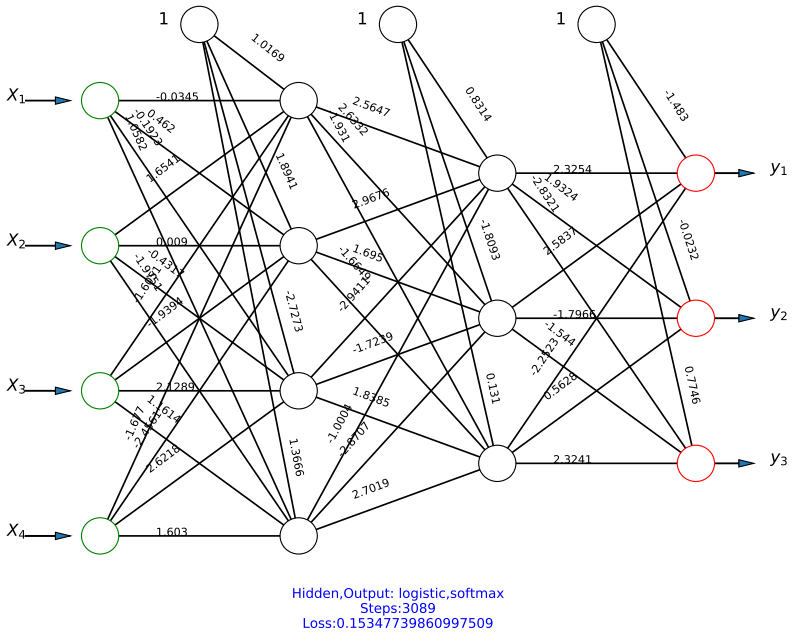
Pytanie: Czy na podstawie otrzymanego wykresu można zasugerować jakąś optymalną wartość obu progów dla tego zbioru przykładów i tej sieci?

Patrząc na Rys. 19 można zaproponować dwie wartości $K1$ (oraz $K0$). Pierwsza wartość to $K1 = 0.6$, ponieważ różnica między procentem poprawnie zaklasyfikowanych, a źle zaklasyfikowanych przykładów jest największa. W porównaniu wyników dla $K1 = 0.6$ z wynikami dla $K1 = 0.5$, to dla lekkiego spadku poprawności, uzyskaliśmy duży spadek popełnianych błędów, co jest dobrym wynikiem.

Druga interesująca wartość parametru to $K1 = 0.9$. Jest to bardzo radykalne podejście, które ma dużą zaletę. W tym przypadku mówimy, że dopiero jak algorytm jest pewny w co najmniej 90%, to wtedy może dokonać klasyfikacji. Dla takiego przypadku, szansa na błędną klasyfikację jest minimalna (co widać na wykresie, gdzie procent źle zaklasyfikowanych oscyluje w okolicach 1%). Zostawia nas to z wieloma przypadkami, które nie mają żadnej klasyfikacji, ale dla takich przypadków można przecież opracować osobny klasyfikator. Dzięki takiemu podejściu nie jesteśmy w stanie zaklasyfikować około 75% danych, ale w pozostałych

25% przypadków jesteśmy pewni poprawnej klasyfikacji.

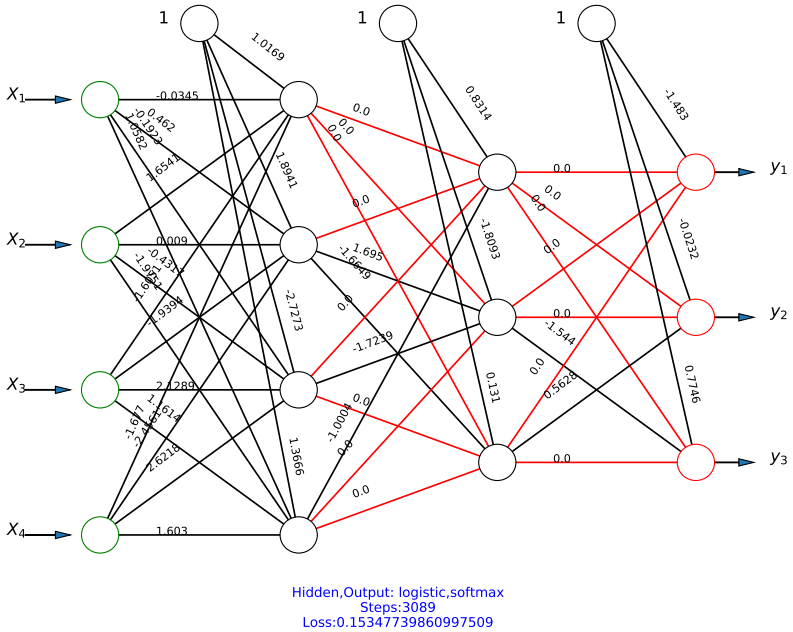
5 Badanie odporności sieci na uszkodzenia



Rysunek 20: Sieć 4–3–3 dla zbioru danych *IRIS*. Sieć została wytrenowana z parametrami *activation*=*logistic*, *random_state*=131803, *max_iter*=5000. Trafność sieci wynosi 100% dla zbioru testowego.

Zbiór danych dla tego zadania to *IRIS*. Zawiera 150 przykładów, z czego 15 z nich zostanie użytych jako zbiór testowy. Tak samo jak w przypadku zadania 4 zbiór został przeskalowany z użyciem *StandardScaler*. Wytrenowana sieć 4–3–3 widoczna jest na Rys. 20. Jej trafność na zbiorze testowym wynosi 100%.

Następnie dla tej sieci zostało wyzerowane 15 wag, zaczynając od wag z najmniejsza wartością absolutną. Oznacza to że usuwane były w pierwszej kolejności wagi najmniej znaczące. Sieć po wyzerowaniu 15 wag jest przedstawiona na Rys. 21.



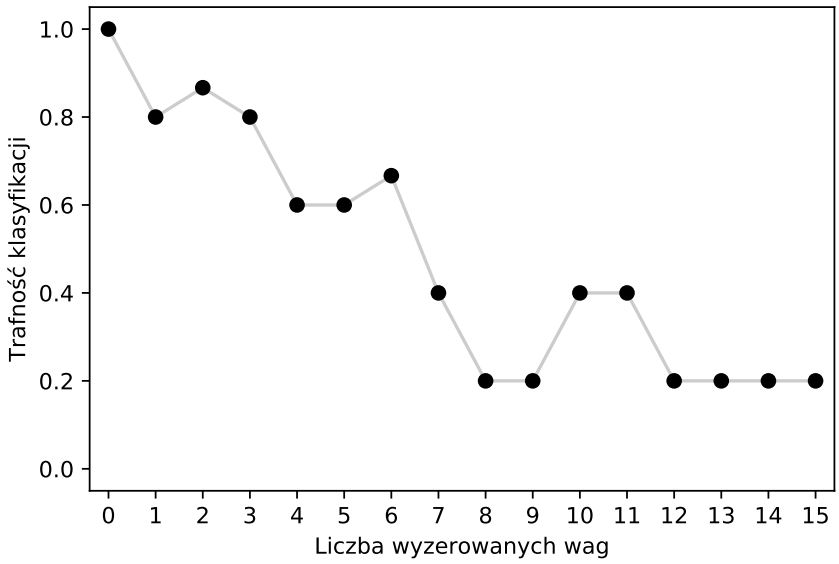
Rysunek 21: Sieć 4–3–3 dla zbioru danych *IRIS* z wyzerowanymi wagami. Wyzerowane wagi są zaznaczone na rysunku jako czerwone połączenia między neuronami.

Pytanie: Czy odporność sieci na uszkodzenia (usunięcia wag) jest wysoka?

Patrząc na Rys. 22 widać, że sieć neuronowa nie posiada odporności na zerowanie wag. Już po usunięciu pierwszej wagi trafność spadła od 20 punktów procentowych.

Uszkodzenie wagi bliżej wejścia, prowadzi do propagacji uszkodzenia na dalsze neurony co może rzutować na poważniejsze zmiany na wyjściu sieci. Można zaobserwować także zjawisko kompensacji błędu, gdzie uszkodzenie kolejnej wagi prowadzi do lepszej klasyfikacji niż przed uszkodzeniem, pod warunkiem, że co najmniej jedna waga została uszkodzona wcześniej.

Podczas testowania, zdarzyło się ziarno losowe, dla którego usunięcie 3 pierwszych wag, nie wpłynęło na trafność klasyfikacji. Taka sytuacja jest bardzo rzadka, dlatego w rzeczywistości sieci są bardzo wrażliwe na wszystkie uszkodzenia. Uszkodzenie wagi propaguje się do neuronu, który następnie wysyła złą wartość do wszystkich neuronów w następnej warstwie. Uzyskujemy efekt lawiny, gdzie mała zmiana prowadzi ostatecznie do poważnych konsekwencji.



Rysunek 22: Wykres trafności klasyfikacji w funkcji liczby wyzerowanych wag. Czarne kropki oznaczają konkretne pomiary, szara linia służy jedynie jako pomoc dla czytelnika.

Warto też wspomnieć o działaniach grupy *Google Brain*, która opracowała metodę na wymuszanie porządnej odpowiedzi od sieci neuronowej [3]. Sieć jest wrażliwa nie tylko na uszkodzenie połączeń między neuronami, ale także na uszkodzenie danych wejściowych.

Pytanie: Czy jesteś w stanie na podstawie tak „zredukowanej" sieci powiedzieć coś o ważności poszczególnych atrybutów opisujących przykłady?

Ponieważ wagi były modyfikowane między warstwami ukrytymi oraz wyjściem, to nie można nic powiedzieć na temat ważności atrybutów. Pomiędzy zaczepami wejściowymi, a pierwszą warstwą ukrytą istnieją już wagi oraz gęste połączenie, które razem powodują, że do poszczególnych neuronów warstwy ukrytej trafiają wartości, które są mieszanką wszystkich atrybutów po trochu.

W takiej sytuacji nie da się nic powiedzieć o ważności atrybutu, bo nie operujemy już na faktycznych atrybutach, tylko na cechach, które sieć sama stworzyła w toku uczenia. Cechy te są mieszanką wszystkich atrybutów po trochu, co tworzy nieinterpretowalną dla człowieka reprezentację wiedzy.

Jedyny sposób, aby sprawdzić ważność atrybutu w sieci neuronowej, to wytrenować sieć z atrybutem i bez niego, a następnie porównać wyniki dla tych sieci.

Pytanie: Czy w konsekwencji takiego „przerzedzenia" sieci jakiego dokonałeś powyżej można usunąć niektóre neurony? Jeśli tak, to jak należy to zrobić?.

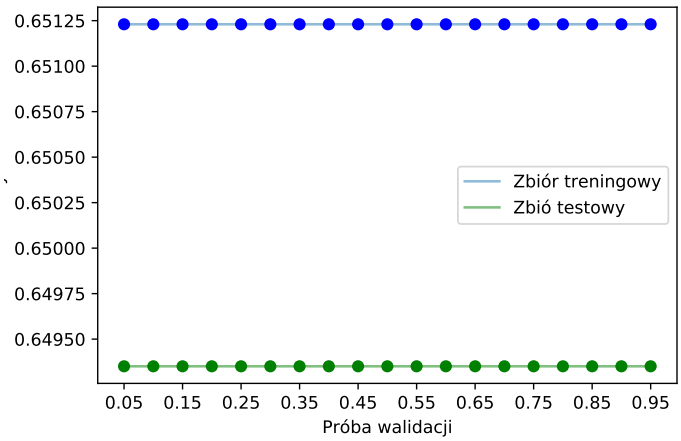
W teorii usunięcie neuronu oznacza wyzerowanie wszystkich wag, które z niego wychodzą. Jednakże nie oznacza to, że wytrenowanie sieci z jednym neuronem mniej w odpowiedniej warstwie da takie same wyniki jak wyzerowanie wag wychodzących neuronu. Jest to spowodowane tym, że uczenie oryginalnej sieci opierało się o wszystkie neurony, dlatego sieć wykorzystała wszystkie neurony przy wyznaczaniu wag. Nauczenie takiej sieci z zredukowaną liczbą neuronów spowoduje wyznaczenie innych wag dla innych połączeń, co ostatecznie będzie propagowało się dalej.

Ciężko jest też usunąć neuron z sieci, bo rzadko się zdarza, żeby wyzerowanie wszystkich wyjściowych wag neuronu nie wpłynęło na trafność klasyfikacji.

Jeśli chcemy usunąć neuron z sieci, to najlepiej jest wytrenować nową sieć o mniejszej liczbie neuronów w odpowiedniej warstwie. Wytrenowanie nowej sieci jest najbezpieczniejszym rozwiązaniem na usunięcie nadmiarowych neuronów, nie rzutującym na możliwości klasyfikacji oryginalnej sieci.

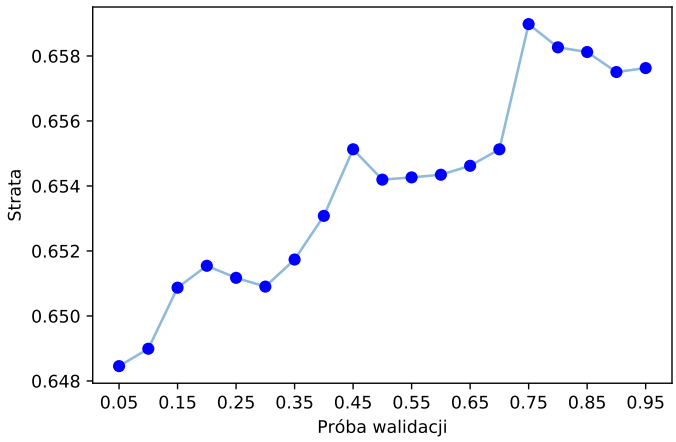
6 Eksperymentalny dobór rozmiaru zbioru walidacyjnego

6.1 Przeprowadzenie eksperymentu uczenia i testowania, zmieniając proporcje pomiędzy zbiorem uczącym i walidującym dla zbioru danych *PIMA*.



Rysunek 23: Wykres przedstawia zmianę trafności klasyfikowania na zbiorze treningowym i testowym dla poszczególnych eksperymentów różniących się parametrem *validation_fraction* (próba walidacji).

Dane zostały podzielone na zbiór treningowy (90% wszystkich przypadków) i testowy (10% wszystkich przypadków). Wykonane zostały eksperymenty gdzie modyfikowany był



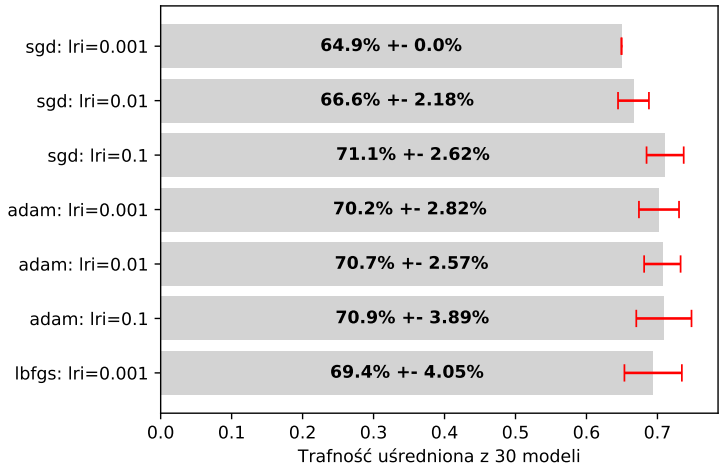
Rysunek 24: Wykres przedstawia funkcję straty dla poszczególnych eksperymentów różniących się parametrem *validation_fraction* (próba walidacji).

parametr *validation_fraction* [7] odpowiadający za proporcję zbioru walidacyjnego. Parametr ten zmieniał się w zakresie od wartości 0.05 do 0.95. Uczenie klasyfikatora odbywało się z maksymalną liczbą iteracji ustawioną na 350. Uczenie modelu ograniczone było też parametrem *early_stopping*, który kończy proces trenowania, gdy ocena trafności na zbiorze walidacyjnym się nie poprawia się przez 10 iteracji.

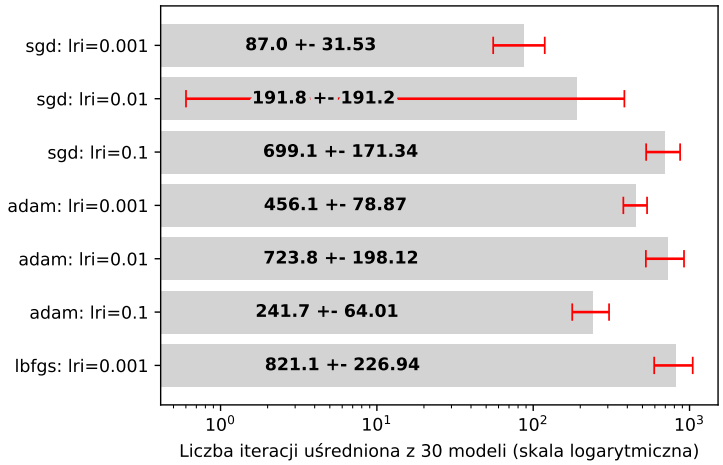
Podczas testu wykonywane były pomiary trafności klasyfikowania i wartości funkcji straty dla kolejnych eksperymentów. Rys. 23 przedstawia wartości trafności mierzonej na zbiorze treningowym jak i testowym. Widać, że trafność klasyfikowania nie zmienia się, dodatkowo każdy pojedynczy eksperyment zakończył się w tej samej iteracji. Zmieniła się jednak funkcja straty pokazana na Rys. 24, rośnie ona wraz ze wzrostem rozmiaru zbioru walidacyjnego w stosunku do malejącego w każdej iteracji zbioru treningowego. W takim wypadku bazując na informacji z Rys. 23 i Rys. 24 najlepszym wyborem byłoby ustawienie parametru *validation_fraction* na wartość 0.05, ponieważ mamy największą pewność, że wagi najlepiej opisują badany problem (najmniejsza wartość funkcji straty), a jednocześnie trafność klasyfikowania na zbiorze treningowym i testowym nie zależy od tego parametru.

7 Porównanie różnych algorytmów uczenia nadzorowanego sieci warstwowych

Eksperymenty zostały przeprowadzone dla poszczególnych konfiguracji sieci neuronowych: dla optymalizatora *sgd* [9] oraz *adam* [1] parametr *learning_rate_init* $\in \{0.001, 0.01, 0.1\}$ [7], a dla optymalizatora *lbfgs* [2] parametr *learning_rate_init* $= 0.001$. Wszystkie testy wykorzystywały tą samą architekturę czyli: 8–4–1. Każdy neuron miał sigmoidalną funkcję aktywacji. Maksymalna liczba iteracji wynosiła 1000, o było wystarczajcie, aby algorytmy był w stanie osiągnąć minimum funkcji straty. Sieci były trenowane 30 razy, z różnymi wartościami parametru *random_state*. Parametr ten był zależny od numeru iteracji. Takie ustawienia pozwoliły na dobre uśrednienie uzyskanych wyników. Każdy eksperyment był także powtórzony dwa razy z drobnymi zmianami parametrów. W drugim eksperymencie włączony został pa-

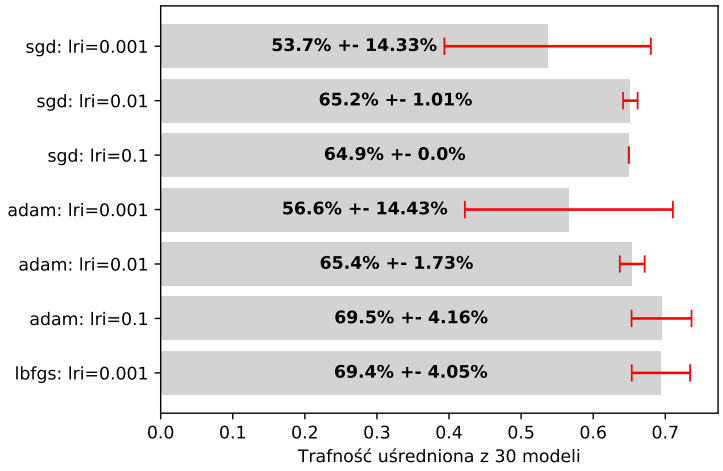


Rysunek 25: Średnia trafność klasyfikacji dla zbioru *PIMA*. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init* [7]. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.

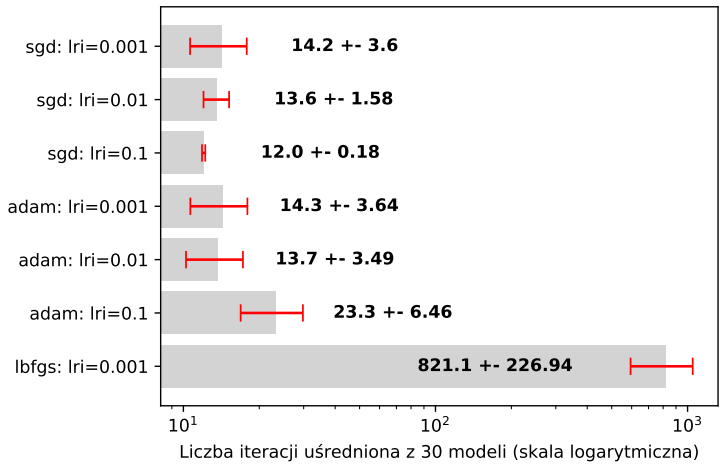


Rysunek 26: Średnia liczba iteracji wymagana do osiągnięcia optimum przez optymalizator dla zbioru *PIMA*. Wartości przedstawione są na skali logarytmicznej. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init*. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.

rametr *early_stopping*, a w trzecim ustawiono parametr *alpha*=0.01. Parameter *alpha* jest odpowiedzialny za regularyzację *L2*. Regularyzacja to dodatkowa kara dla siei, która jest metodą na zmniejszenie przeuczenia. Pozostałe parametry pozostawione były na ustawieniach domyślnych. Testy były przeprowadzone na zbiorze *PIMA*.

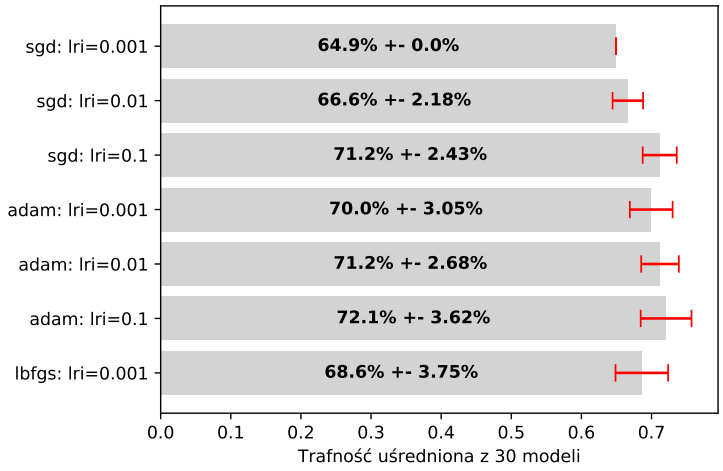


Rysunek 27: Średnia trafność klasyfikacji dla zbioru *PIMA* przy użyciu parametru *early_stopping=True*. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init*. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.

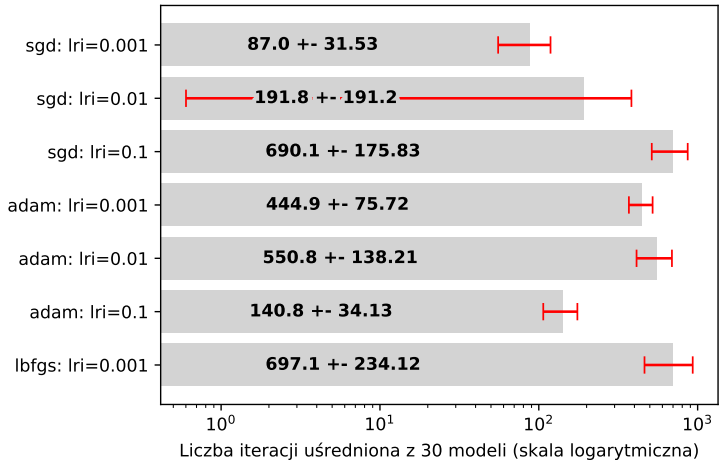


Rysunek 28: Średnia liczba iteracji wymagana do osiągnięcia optimum przez optymalizator dla zbioru *PIMA* przy użyciu parametru *early_stopping=True*. Wartości przedstawione są na skali logarytmicznej. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init*. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.

Wyniki eksperymentów są przedstawione na wykresach Rys. 25, Rys. 26; dla wariantu z *early_stopping* na Rys. 27, Rys. 28; dla wariantu z *alpha*=0.01 na Rys. 29 i Rys. 30.



Rysunek 29: Średnia trafność klasyfikacji dla zbioru *PIMA* przy użyciu parametru *alpha*=0.01. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init*. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.



Rysunek 30: Średnia liczba iteracji wymagana do osiągnięcia optimum przez optymalizator dla zbioru *PIMA* przy użyciu parametru *alpha*=0.01. Wartości przedstawione są na skali logarytmicznej. Parametry użyte do definicji modelu znajdują się po lewej stronie. Jest to kolejno nazwa zastosowanego optymalizatora oraz wartość użytego parametry *learning_rate_init*. Kolejne modele miały ustawione parametry *random_state* jako numer pozycji klasyfikatora w wynikowej tablicy.

Porównanie wyników

Patrząc na wykresy reprezentujące trafność klasyfikacji, czyli Rys. 25, Rys. 27 i Rys. 29 można zaobserwować, że dla algorytmów *sgd* oraz *adam* regularyzacja ustawiona na 0.01 raczej

poprawiała wyniki, a na pewno ich nie pogorszyła. Natomiast z użyciem regularyzacji ucierpiał algorytm *lbfgs*. Parametr *early_stopping* zawsze pogarszał wyniki i to całkiem w dużym stopniu. Jego jedyną przewagą jest bardzo szybkie trenowanie sieci co pokazuje Rys. 28. Patrząc na przykład *sgd* z parametrem *learning_rate_init*=0.1 trenowanie sieci było o 58 razy szybsze dla parametru *early_stopping*. Należy wspomnieć, że algorytm *lbfgs* nie używa parametru *early_stopping*, dlatego nie ma różnicy w wynikach między Rys. 26, a Rys. 28 oraz między Rys. 25, a Rys. 27 dla algorytmu *lbfgs*.

Pytanie pojawia się jaki algorytm najlepiej się spisuje. Niestety nie jest to jednoznaczna odpowiedź. W przypadku wykresu Rys. 25 widać, że najlepszą trafność uzyskał *sgd*, ale *adam* osiągnął dobre wyniki bez względu na parametr *learning_rate_init*. W przypadku Rys. 27 *adam* w porównaniu do *sgd* poradził sobie lepiej przy włączonym parametrze *early_stopping*. Mówiąc o optymalizatorze *lbfgs* jest to dobry wybór, który plasuje się miedzy pozostałymi. Natomiast dla pokazanych wykresów można powiedzieć że *adam* jest nie najlepszym wyborem, chociaż należy pamiętać, że *sgd* też może dostarczyć dobre wyniki. Wszystko sprowadza się także do architektury sieci, wielkości zbioru danych, jakości tych danych oraz złożoności problemu. W takim wypadku lepiej nie mówić, że jeden algorytm jest lepszy od drugiego, a raczej należy powiedzieć, że wybór *adam* optymalny w tym konkretnym przypadku.

Można też patrzeć na kryterium ilości iteracji w celu porównania klasyfikatorów. W takim wypadku występuje ciekawa obserwacja. Dla większych wartości parametru *learning_rate_init* algorytm *sgd* wykonuje coraz więcej iteracji, natomiast dla algorytmu *adam* sytuacja jest prawie odwrotna (bo dla środkowej wartości, występuje anomalia). Jeszcze więcej zamieszania wprowadza parametr *early_stopping* (Rys. 28), gdzie to co jest napisane wyżej, jest odwrócone. Dlatego jeśli mówimy o najlepszym algorytmie ze względu na ilość potrzebnych iteracji to najgorszym wyborem jest *lbfgs*, a dla pozostałych optymalizatorów to zależy od parametru *learning_rate_init*. Nie można tutaj doradzić uniwersalnego algorytmu.

8 Dla chętnych: inne sieci warstwowe – sieci z jednostkami o symetrii kołowej (RBF)

W przypadku sieci *RBF* funkcja *PSP* (funkcja potencjału postsynaptycznego [11]) jest jądrem Gaussa. Jest to dwuargumentowa funkcja kowariancji licząca odległość otrzymanej od środka grupy wyznaczonej metodą k–średnich. Funkcja jest liczona dla każdej grupy (każdego centrum) i każdego sygnału wejściowego. W ten sposób uzyskiwana jest macierz o rozmiarach $l_w \times l_c$, gdzie l_w to liczba wejść do neuronu, a l_c to liczba centrów. W efekcie funkcja *PSP* dla sieci *RBF* zwraca macierz. Sieć *RBF* ma liniową funkcję aktywacji, czyli to co dostaje na wejściu, przepisuje na wyjście. W tym wypadku jest to cała macierz zwracana przez funkcję *PSP*.

Jest zupełnie inna filozofia w porównaniu do tradycyjnych sieci, gdzie funkcja *PSP* jest najczęściej iloczynem skalarnym wektora wejść z wektorem wag. Taka funkcja zwraca ostatecznie tylko i wyłącznie pojedynczą liczbę. Ta liczba trafia ostatecznie do funkcji aktywacji, najczęściej są to funkcje *ReLU*, *sigmoid*, *softmax* itp. Są to funkcje, które na wejściu przyjmują jeden argument i na wyjściu zwracają jedną liczbę.

Niestety o ile udało się wytrenować sieć *RBF* dla zbioru *IRIS*, nie jesteśmy w stanie powiedzieć, co sieć faktycznie zwraca. Dla problemu *IRIS* występują 3 klasy decyzyjne i mimo definiowania, że sieć powinna mieć 3 neurony wyjściowe, to po treningu sieć zostawała z tylko jednym neuronem wyjściowym. Pomysł był taki, że można spróbować zastosować metodę *OvA*, aby nauczyć 3 sieci dla każdej klasy. Jednakże w rzeczywistości mamy wrażenie, że sieć *RBF* jest siecią regresyjną, a nie klasyfikacyjną, bo nie zwraca ona prawdopodobieństw (niektóre wartości zwracane przez sieć były znacznie większe niż 1, np.: 2.7 i 3.1).

Jednakże na podstawie uruchomionych eksperymentów można powiedzieć, że sieci z jednostkami sigmoidalnymi zawsze są w zakresie od 0 do 1, podczas gdy zaobserwowane wartości jądra Gaussa dla sieci *RBF* nigdy nie przekroczyły wartości 1.

Mówiąc o promieniach jądra Gaussowskiego (inaczej ten parametr jest znany jako *beta* lub *lenght-scale*) trzeba zastanowić się jak zachowuje się wykres funkcji kowariancji. Dla wartości parametru *beta* bliskich zera, funkcja ta będzie miała wysoki „czubek" oraz bardzo ostre boki, ale mały promień. To może prowadzić w sieci do przetrenowania, gdyż taka funkcja słabiej będzie generalizować. Natomiast kiedy parametr *beta* jest daleko od wartości zero, to wtedy funkcja będzie „rozlana" po płaszczyźnie o łagodnych zboczach i bardzo niskim „czubku". Taki kształt funkcji może faworyzować generalizację, ponieważ pokrywa większy obszar, jednakże dla tego obszaru zwraca mniejsze wartości, niż w porównaniu do małych wartości parametru *beta*.

9 Uzyskiwanie jak najwyższej trafności

W celu zapewnienia powtarzalności eksperymentów, wszystkie algorytmy były trenowane przy użyciu *random_state*=1. Jest to bardzo konieczne, ponieważ losowa inicjalizacja wag, ma bardzo duże znaczenie w przypadku sieci neuronowych, zwłaszcza dla takich małych zbiorów danych jakimi zajmujemy się tutaj.

Zbiór danych wybrany do eksperymentów to *PIMA*. Został podzielony na zbiór testowy i treningowy. Dodatkowo na podstawie zbioru treningowego wytrenowano obiekt *StandardScaler* [8] użyty później do przeskalowania zbioru testowego. Wszystkie klasyfikatory wytrenowane były za pomocą przeskalowanego zbioru treningowego.

Sieci neuronowe mają bardzo wiele odmiennych hiper–parametrów, dlatego zdecydowałem, że najlepiej będzie wytrenować trzy osobne sieci i na każdej uzyskać jak najlepsze wyniki. Następnie wybrać najlepszą z nich. Sieci podzieliłem na trzy rodziny, ze względu na zastosowany optymalizator. Dostępne optymalizatory są to *lbfgs*, *sgd* oraz *adam*. Każdy z optymalizatorów ma inny zbiór dostępnych parametrów.

9.1 Optymalizator *LBFGS* [4]

Z dokumentacji *SciKit–Learn* wynika, że optymaliator *LBFGS* jest bardzo wydajny dla małych zbiorów danych. Ma to potwierdzenie w eksperymentach, ponieważ sieć była najszybciej trenowana właśnie z tym optymalizatorem.

Dla *LBFGS* sprawdzone zostały różne architektury dwu i jedno warstwowe. W przypadku architektur dwuwarstwowych następowało ciągłe przeuczanie. Najlepsza natomiast okazała się architektura jednowarstwowa o 4 neuronach, czyli 4–1.

W przypadku funkcji aktywacji najlepsze rezultaty dawała funkcja *ReLU* (w porównaniu do funkcji sigmoidalnej – *logistic*).

W przypadku regularyzacji najmniejszy błąd na zbiorze testowym uzyskała sieć z parametrem o wartości 0.0015. Testowane były różne wartości parametru *alpha* np.: 0.0001, 0.001, 0.0005 i 0.01.

Maksymalna liczba iteracji/epok wynosi 5000. Dzięki temu algorytm jest wstanie osiągnąć minimum funkcji straty.

Ostatecznie wytrenowana sieć uzyskała trafność równą 77.9%.

9.2 Optymalizator *ADAM* [1]

W przypadku optymalizatora *adam* testowano różne architektury podobnie jak poprzednio. Okazało się, że najlepsze rezultaty daje sieć o architekturze 2–1, czyli sieć jednowarstwowa z

dwoma neuronami.

Optymalizator *adam* udostępnia dwa parametry: *beta_1* oraz *beta_2*. Najlepsze ustawienie dla nich to *beta_1*=0.01 oraz *beta_2*=0.99999. Parametr *beta_1* był testowany dla różnych wartości w zakresie od 0 do 0.9, a *beta_2* w zakresie od 0.95 do prawie 1. Te parametry wpływają na *learning_rate* liczony na podstawie średniej drugiego momentu gradientu (wariancji gradientu). Te parametry określają stosunek „rozpadu" (ang. *decay*) liczonych średnich. Domyślne parametry były równe *beta_1*=0.9 oraz *beta_2*=0.999, co oznacza, że obecnie dobrane parametry są bardzo różne od domyślnych.

W przypadku parametru *learning_rate_init*, wybrano wartość 0.0055. Inne testowane wartości to 0.01, 0.005, 0.001, 0.0005, 0.0001. Parametr ten służy do definiowana początkowej wartości kroku po gradiencie. W przypadku zbyt małych wartości, algorytm będzie bardzo wolno schodził po gradiencie, przez co może dojść do limitu iteracji szybciej, niż minimum funkcji straty. Natomiast zbyt duża wartość tego parametru może prowadzić do chaotycznego przemieszczania się po gradiencie, „skakania" po powierzchni funkcji straty. W najgorszym wypadu taki parametr może spowodować, że skoki będą na tyle duże, że będą oddalać się od globalnego minimum.

Maksymalna liczba iteracji została ustawiona na 5000. W przypadku innych parametrów zostały wybrane wartości domyślne, ponieważ nie było różnicy w trafności, albo trafność się pogarszała.

Ostateczna sieć uzyskała trafność równą 79.2%.

9.3 Optymalizator *SGD*

Ostatnia testowana sieć była oparta o algorytm *SGD*, czyli stochastyczny spadek wzdłuż gradientu. W przypadku każdej sieci typu *feed–forward* funkcja starty przybiera wypukły kształt, dlatego znalezienie minimum jest traktowane jako spadek do minimum.

Architektura była testowana podobnie jak przy poprzednich sieciach, ale tutaj okazało się, że najlepsze wyniki daje sieć w architekturze 8–3–1, czyli sieć dwuwarstwowa. Testowane były również sieci o trzech warstwach ukrytych, ale pogarszały wyniki na zbiorze testowym.

Początkowy *learning_rate* ustawiono na wartość 0.0055, czyli identycznie jak dla optymalizatora *adam*. W testach ta wartość dawała najlepsze rezultaty.

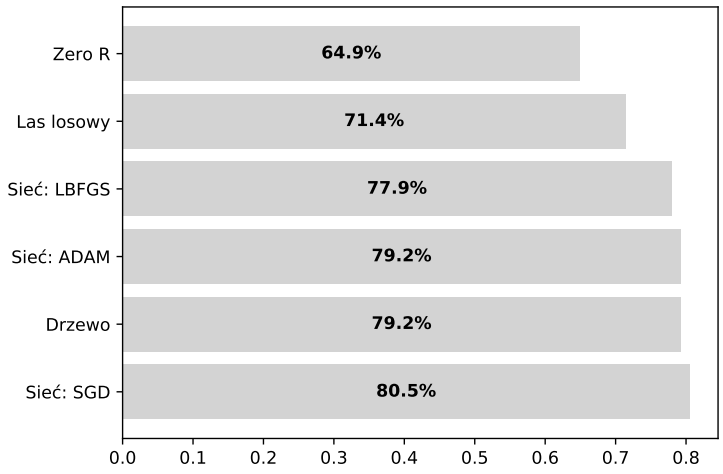
Testowane były także różne warunki stopu i inne ustawienia, jednakże nic poza domyślne parametry nie dało lepszych rezultatów. Wykonane były też eksperymenty z adaptatywnym *learning_rate*, jednakże pogarszało to wyniki.

Ostatecznie sieć uzyskała trafność: 80.5%.

9.4 Porównanie wyników

Podsumowanie wyników zawiera wszystkie najlepsze wytrenowane sieci, a także drzewo decyzyjne, las losowy oraz klasyfikator *Zero R*. Rys. 31 zawiera wszystkie wyniki, gdzie najlepszy wynik jest na dole wykresu.

Jak widać sieci na pierwszy rzut oka spisały się bardzo dobrze dla tego problemu. Należy jednak wziąć pod uwagę ile należało dobrać parametrów, aby takie wyniki uzyskać, podczas gdy drugie miejsce w rankingu zajmuje drzewo decyzyjne z domyślnymi parametrami. W takim kontekście okazuje się, że sieć neuronowa nie ma znaczącej przewagi nad drzewem, co bierze się z natury problemu. Zbiór danych *PIMA* jest w praktyce mały jeśli mówimy o sieciach neuronowych. Dla małych zbiorów danych lepszym wyborem są klasyczne metody uczenia maszynowego, natomiast sieci neuronowe lepiej działają dla dużych zbiorów danych i bardziej złożonych problemów.



Rysunek 31: Porównanie trafności klasyfikatorów. Trafność pokazana procentowo na każdej belce. Jako punkt odniesienia użyto klasyfikator *Zero R*, który wybiera klasę większościową, w tym wypadku klasyfikuje wszystko jako klasę 0.

Literatura

[1] Vitaly Bushaev. Adam — latest trends in deep learning optimization, 2018.

[2] Wikipedia contributors. Limited-memory bfgs — Wikipedia, the free encyclopedia, 2020.

[3] Gamaleldin F. Elsayed, Shreya Shankar, Brian Cheung, Nicolas Papernot, Alex Kurakin, Ian Goodfellow, and Jascha Sohl-Dickstein. Adversarial examples that fool both computer vision and time-limited humans, 2018.

[4] S. Wright Jorge Nocedal. *Numerical Optimization*. Morgan Kaufmann Publishers, Inc., 2nd edition, 2006.

[5] Developers of Jupyter. Jupyter notebook documentation, 2015.

[6] Developers of Python3. Python3 documentation, 2020.

[7] Developers of Scikit-learn. Mlpclassifier documentation, 2019.

[8] Developers of Scikit-learn. Standardscaler documentation, 2019.

[9] Developers of Scikit-learn. Stochastic gradient descent documentation, 2019.

[10] Developers of Scikit-learn. Scikit–learn documentation, 2020.

[11] Inc. StatSoft. Statsoft – electronic statistics textbook, 2020.