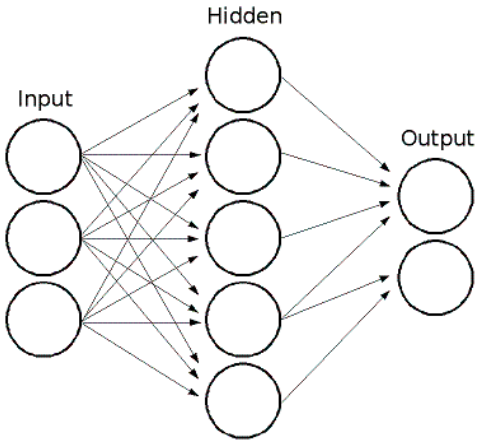
ĆWICZENIE 2 I 3
Płytka sieć jednokierunkowa do klasyfikacji cyfr ze zbioru MNIST

Joanna Chlebus
242505
Październik 2019

1 Plan eksperymentów

Celem przeprowadzonych eksperymentów było zapoznanie się z architekturą **płytkiej sieci neuronowej MLP** (*Multilayer Perceptron*), zasadą jej uczenia za pomocą algorytmu propagacji wstecznej oraz zbadanie wpływu parametrów odgrywających istotną rolę w procedurze uczącej. W ćwiczeniu skupiono się na badaniu zależności uczenia od wielkości paczki uczącej, sposobu inicjalizacji wag początkowych każdej warstwy, liczby neuronów w warstwie ukrytej oraz zastosowanej funkcji przejścia. Przeprowadzono również eksperymenty dotyczące różnych sposobów optymalizacji wielkości współczynnika uczenia w trakcie trwania uczenia sieci oraz wpływ funkcji straty na szybkość uczenia.



Rysunek 1: Model sieci wielowarstwowej z jedną warstwą ukrytą (*Hidden*)

Do wykonania eksperymentów przygotowano program, który symulował działanie sieci neuronowej. Zaimplementowano w nim architekturę MLP składającą się z warstw *Fully Connected* z jedną warstwą ukrytą oraz wykorzystywane w eksperymentach funkcje aktywacji. Algorytm propagacji wstecznej napisano, korzystając z **reguły łańcuchowej** - gradient funkcji straty obliczano w odniesieniu do każdej wagi poprzez iteracyjne domnażanie kolejno wyliczanych gradientów z pojedynczej warstwy, zaczynając od ostatniej. Aktualizacja wag odbywała się po paczce (***batch***), czyli po podaniu pewnej części wzorców z danych treningowych. Po każdej epoce uczenia (tzn. po zobaczeniu przez sieć całego zbioru treningowego) liczono i zapamiętywano wartość **funkcji kosztu** oraz **skuteczność** uczenia dla ciągu treningowego i walidacyjnego. Jako miarę skuteczności uczenia przyjęto liczbę poprawnych predykcji sieci na liczbę wszystkich wzorców z danego ciągu danych. Warunkiem stopu

procedury uczącej była zadana z góry maksymalna liczba epok. Przyjęto następujące **domyślne wartości** parametrów modelu sieci:

- liczba neuronów w warstwie ukrytej - 500
- funkcja straty - entropia krzyżowa połączona z softmaxem
- metoda inicjalizacji wag początkowych - Xavier
- funkcja aktywacji - sigmoidalna
- współczynnik uczenia - 0.01
- optymalizator współczynnika uczenia - SGD (*Stochastic Gradient Descent*

Domyślne parametry dla procedury uczenia:

- liczba epok - 30
- wielkość batcha - 50

W każdym z eksperymentów zmieniano jeden lub kilka powyższych parametrów, reszta ustalonych wartości pozostawała niezmienna.

2 Opis aplikacji wykorzystywanej do badań

Symulację zachowania neuronów prostych zaimplementowano w języku Python w środowisku PyCharm. Skorzystano m.in. z biblioteki *numpy* ułatwiającej operacje macierzowe oraz *sklearn*. Struktura projektu przedstawia się następująco:

```
neural-net
├── experiments
│   ├── experiments.py
│   └── run_training.py
├── net
│   ├── layers
│   │   ├── activations.py
│   │   ├── dense.py
│   │   ├── input.py
│   │   └── layer.py
│   ├── model
│   │   ├── architectures.py
│   │   └── model.py
│   ├── batcher.py
│   ├── callbacks.py
│   ├── initializers.py
│   ├── losses.py
│   ├── metrics.py
│   ├── optimizers.py
│   └── loggers.py
├── utils.py
└── training.py
```

W pakiecie `layers` zaimplementowano poszczególne warstwy sieci neuronowej m.in. z metodami przejścia wprzód (***forward***) oraz wstecznej propagacji (***backward***). Tam też znajdują się funkcje aktywacji. W pakiecie `model` można znaleźć klasę modelu umożliwiającą sekwencyjne dokładanie warstw i oferującą zestaw licznych metod potrzebnych do treningu, predykcji oraz zapisywania wag i parametrów uczonej sieci. Plik `architectures.py` zawiera określone już architekury sieci neuronowej, tutaj jedną architekturę składającą się z warstwy wejściowej, ukrytej i wyjściowej. W pliku `batcher.py` napisana jest klasa, której obiekt odpowiedzialny jest za podawanie danych paczkami. Dalej zaimplementowano odpowiednio:

- `callbacks.py` - klasy, których funkcjonalność wykonywana jest w zadanym momencie procedury uczącej

- `initializers.py` - różne metody inicjaliacji wag początkowych warstw
- `losses.py` - funkcje straty
- `metrics.py` - metryki oceniające jakość wyuczonej sieci, dla potrzeb tego ćwiczenia wykorzystano tylko jedną metrykę - skuteczność predykcji wzorców
- `optimizers.py` - optymalizatory współczynnika uczenia
- `loggers.py` - klasy odpowiedzialne za logowanie danych, które chcemy zapisywać na bieżąco w trakcie uczenia np. w celu późniejszego rysowania wykresów czy oceny jakości treningu modelu
- `utils.py` - zawiera metody pomocnicze do ładowania danych oraz rysowania wykresów
- `training.py` - klasa odpowiedzialna za procedurę uczenia modelu sieci

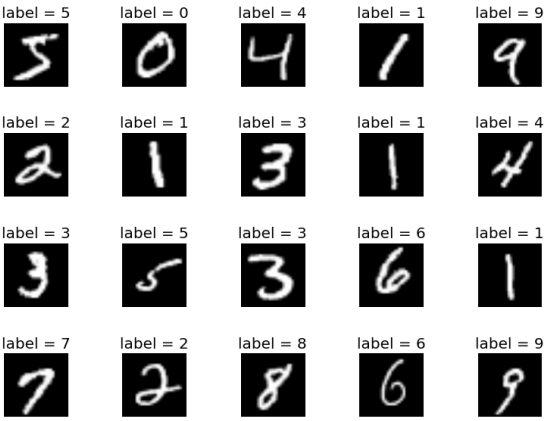
W pakiecie `experiments` znajdują się pliki zawierające skrypty wykonawcze procesu uczenia oraz eksperymentów.

3 Charakterystyka zbiorów danych użytych do badań

Proces uczenia sieci neuronowej wymaga podziału danych na 3 zbiory:

1. ciąg treningowy - służący do wyuczenia wag sieci
2. ciąg walidacyjny - służący do wyboru hiperparametrów oraz nadzorowania jakości uczenia modelu, pokazywany sieci co pewną liczbę iteracji aktualizacji wag (zwykle po pełnej epoce)
3. ciąg testowy - używany na samym końcu do oceny wyuczonego modelu

W ćwiczeniu użyto zbiór danych MNIST składający się z ręcznie pisanych cyfr od 0 do 9 (10 klas). Zawiera on 60 000 **wzorców treningowych** oraz 10 000 **testowych**. Wzorcem jest tu zdjęcie cyfry o wymiarach 28x28 pikseli, przedstawione jako spłaszczony wektor wymiaru 784x1, wraz z odpowiadającą mu etykietą klasy.
Dane sklonowano z repozytorium książki M.Nielsena zostawionej pod linkiem na githubie `https://github.com/mnielsen/neural-networks-and-deep-learning.git`. Z pobranego ciągu treningowego wyłączono 10 000 wzorców na **ciąg walidacyjny**, więc ciąg treningowy zawierał w tym ćwiczeniu 50 000 wzorców. Podział danych na trzy zbiory został wykonany raz i pozostawały one niezmienne co do zawartości w trakcie wszystkich eksperymentów w celu uniknięcia wpływu zmiany ciągów danych na uczenie sieci.



Rysunek 2: Wizualizacja danych MNIST posiadające 10 klas cyfr

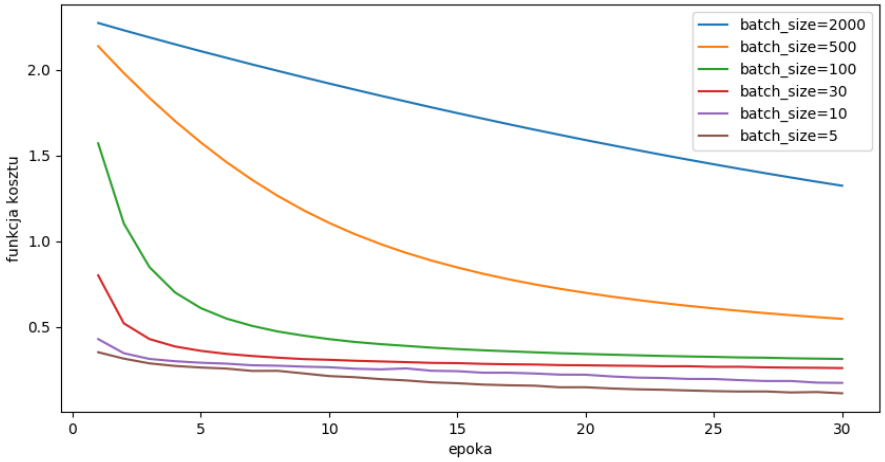
4 Eksperyment 1. Wpływ wielkości paczki danych na wyuczenie sieci

Założenia:

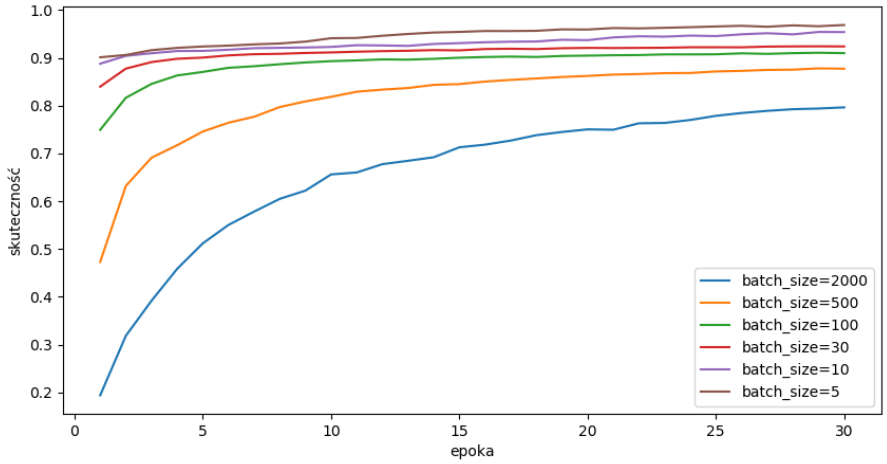
- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - 30 epok
- zmianom ulegała wielkość paczki danych treningowych kolejno ze zbioru {5, 10, 30, 100, 500, 2000}

Przebieg eksperymentu:
Zbadano wpływ wielkości paczki danych treningowych poprzez uruchomienie procedury uczącej modelu sieci raz dla każdej wartości wielkości paczki przy zachowaniu stałych wartości pozostałych parametrów uczenia. Zapamiętywano wartość funkcji kosztu oraz skuteczność dla ciągu walidacyjnego po każdej epoce uczenia w celach porównawczych uczenia sieci dla poszczególnych wielkości badanego parametru. Aby lepiej zobrazować różnice w aktualizacji wag dla poszczególnej wartości wielkości batcha zbadano zmiany funkcji straty po każdym batchu na ciągu treningowych w ciągu trwania pierwszej epoki uczenia. Zmierzono również czas trwania uczenia w sekundach dla każdego wywołania procedury uczącej.

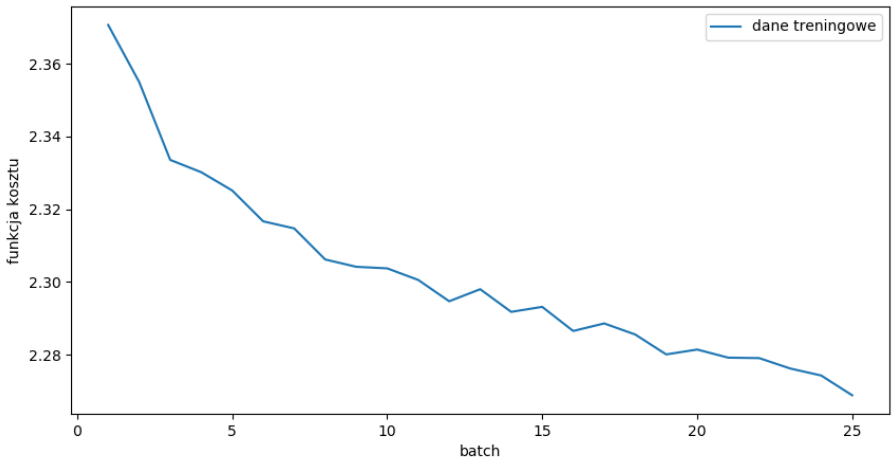
Wyniki:



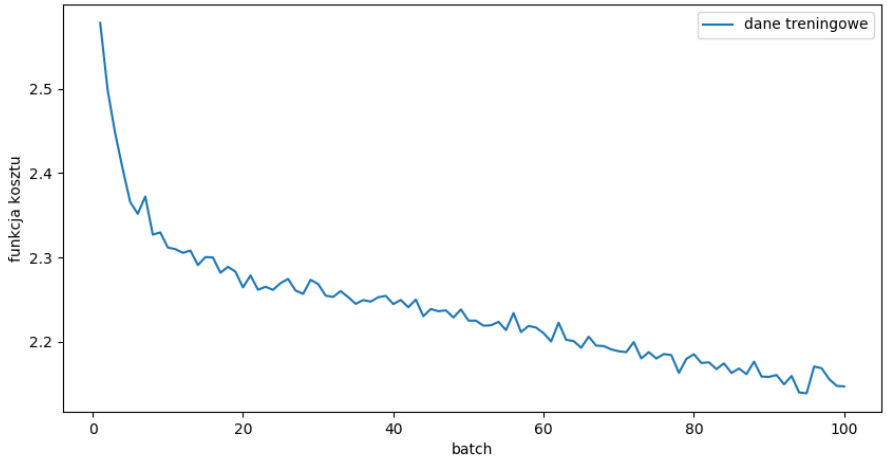
Rysunek 3: Wartość funkcji straty na ciągu walidacyjnym po każdej epoce dla różnego wielkości paczki



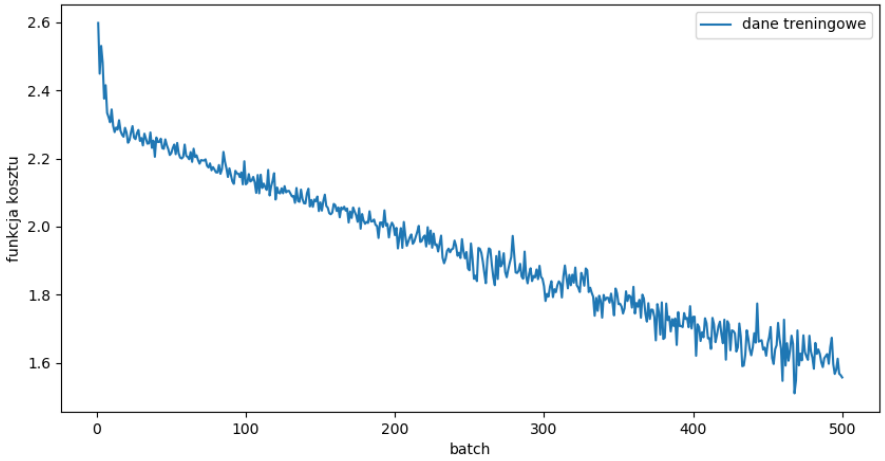
Rysunek 4: Skuteczność na ciągu walidacyjnym po każdej epoce dla różnej wielkości paczki



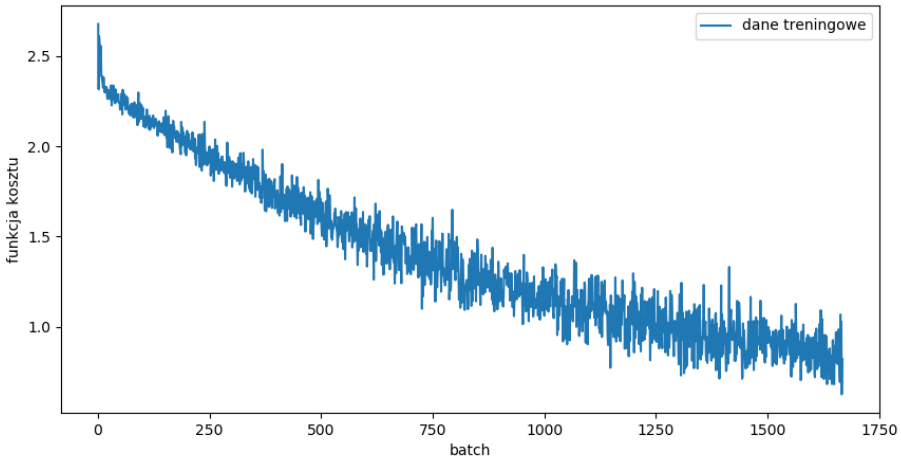
Rysunek 5: Wartość funkcji straty na ciągu treningowym po paczkach wielkości **2000** w ciągu pierwszej epoki uczenia



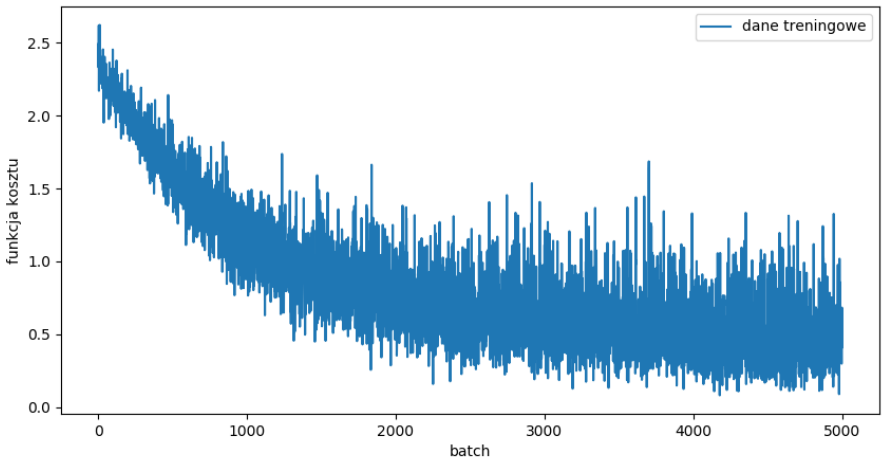
Rysunek 6: Wartość funkcji straty na ciągu treningowym po paczkach wielkości **500** w ciągu pierwszej epoki uczenia



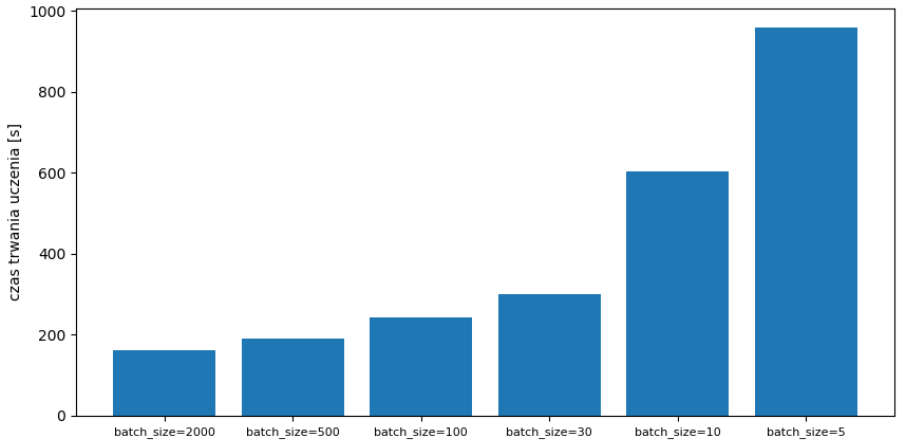
Rysunek 7: Wartość funkcji straty na ciągu treningowym po paczkach wielkości **100** w ciągu pierwszej epoki uczenia



Rysunek 8: Wartość funkcji straty na ciągu treningowym po paczkach wielkości **30** w ciągu pierwszej epoki uczenia



Rysunek 9: Wartość funkcji straty na ciągu treningowym po paczkach wielkości **10** w ciągu pierwszej epoki uczenia



Rysunek 10: Czas trwania uczenia sieci dla różnych wielkości paczek danych w ciągu 30 epok

Komentarz:
Można zauważyć z **Rysunku 3**, że wartość funkcji kosztu na ciągu walidacyjnym spada najmocniej w ciągu pierwszych epok uczenia. Jest to spowodowane tym, że początkowa wartość straty zależy głównie od liczby klas dla wzorców, gdyż początkowe wagi przyjmują wartości losowe. Zachodząca więc w tym czasie predykcja jest również losowa - wagi nie zdążyły jeszcze się wyuczyć rozpoznawania wzorców. Spadek funkcji kosztu skutkuje wzrostem skuteczności klasyfikacji sieci, co obserwujemy na **Rysunku 4**. W początkowych epokach uczenia skuteczność szybko rośnie, następnie jej wartość się stabilizuje i wzrasta powoli.
Im mniejsza wielkość paczki, tym tempo spadku funkcji straty jest szybsze - osiąga ona swoje minimum po mniejszej liczbie epok w porównaniu do uczenia na większych paczkach. Przekłada się to na szybszy wzrost skuteczności na ciągu walidacyjnym dla małych paczek. Dla mniejszego batcha aktualizacja wag w ciągu jednej epoki zachodzi częściej, dzięki czemu sieć się szybciej uczy i wcześniej uzyskuje skuteczność, przy której dalszy jej wzrost się stabilizuje. Warto jednak zauważyć, że im mniejszy batch, tym sieć potrzebuje coraz to dłuższego czasu, aby się wyuczyć, co jest widoczne na **Rysunku 10**.
Na wykresach od **5** do **9** pokazana jest zmiana wartości funkcji straty w ciągu pierwszej epoki uczenia. Mniejsza paczka danych daje częstsze aktualizacje, ale wartość funkcji straty zmienia się wtedy gwałtowniej w zakresie jednego batcha. Natomiast w trakcie uczenia na większych paczkach funkcja straty spada w łagodniejszy sposób, ale jej aktualizacje zachodzą rzadziej.
Po analizie wyników rozsądnym wydaje się dobranie w tym przypadku paczki wielkości 30, ponieważ czas uczenia nie jest dla niej tak długi jak dla batcha równego 5 lub 10, a jednocześnie po małej liczbie epok osiągana jest wysoka skuteczność na ciągu walidacyjnym.

5 Wpływ inicjalizacji wag na wyuczenie sieci

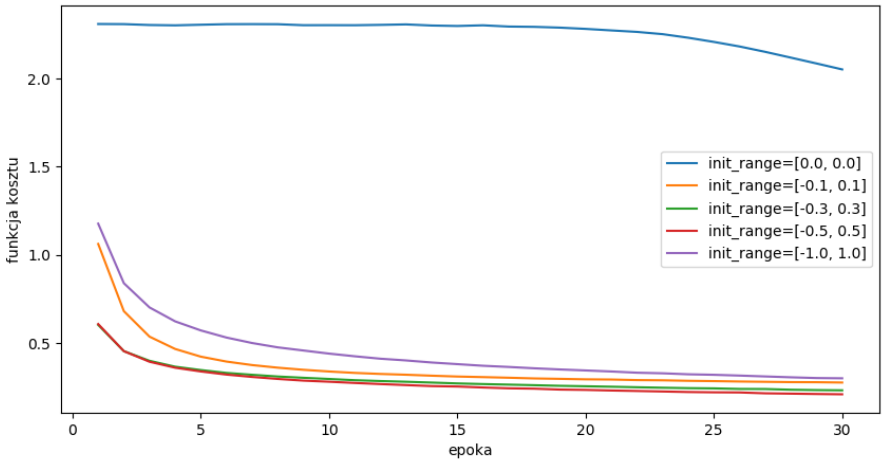
5.1 Eksperyment 2. Zbadanie wpływu zakresu losowanych wag początkowych na proces uczenia

Założenia:

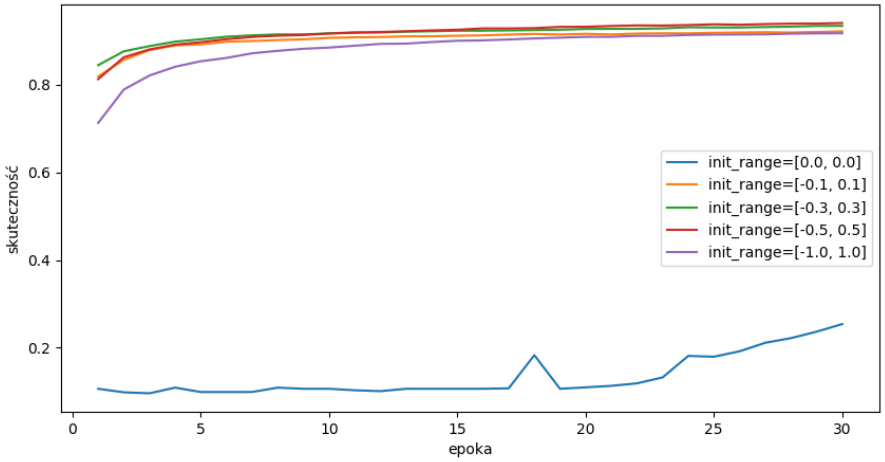
- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - 30 epok
- zmianom ulegał zakres inicjalizacji wag początkowych sieci kolejno z następujących przedziałów: {[0.0, 0.0], [-0.1, 0.1], [-0.3, 0.3], [-0.5, 0.5], [-1.0, 1.0]}, wagi losowano z rozkładu jednorodnego

Przebieg eksperymentu:
Przeprowadzono badania wpływu przedziału losowania wag początkowych warstw sieci neuronowej na proces jej uczenia. Procedurę uczącą uruchamiano kolejno dla każdego z zadanych w założeniach przedziałów wag. Ich inicjalizacja odbywała się z rozkładu jednorodnego. Uczenie modelu odbywało się do osiągnięcia warunku stopu, którym była liczba epok równa 30. Wpływ zakresu inicjalizacji wag porównano poprzez wizualizację na wykresach kolejnych wartości funkcji straty oraz skuteczności co epokę na ciągu walidacyjnym.

Wyniki:



Rysunek 11: Wartość funkcji straty na ciągu walidacyjnym po każdej epoce dla różnych zakresów losowania wag początkowych



Rysunek 12: Skuteczność na ciągu walidacyjnym po każdej epoce dla różnych zakresów losowania wag początkowych

Tabela 1: Skuteczność na ciągu **testowym** dla różnych zakresów inicjalizacji wag

ciąg testowy					
przedział losowania wag	[0.0, 0.0]	[-0.1, 0.1]	[-0.3, 0.3]	[-0.5, 0.5]	[-1.0, 1.0]
skuteczność	25.36%	91.88%	93.12%	93.72%	91.77%

Komentarz:
Wartość funkcji kosztu dla inicjalizacji wszystkich wag na zero (z przedziału [0.0, 0.0]) jest wysoka w porównaniu z innymi przedziałami inicjalizacji, a konsekwentnie też wartość skuteczności na ciagu walidacyjnym dla tych wag jest niska i utrzymująca się na w miarę stałym poziomie. Dodatkowo skuteczność na ciągu testowym dla zerowych wag po 30 epokach była bardzo mała, bo niecałe 30%. Inicjalizacja wszystkich wag na zero nie ma sensu, bo model nie uczy się. Tak dobrane początkowe wagi zero sprawiają, że z każdej warstwy sieci wychodzą takie same wartości, a aktualizacja wag zachodzi o tę samą wartość, bo wyliczane gradienty mają jednolite wartości. Brak więc źródła asymetryczności pomiędzy wagami poszczególnych neuronów.
Dla pozostałych przedziałów losowania wag wraz ze zwiększaniem się ich bezwzględnej wartości krańcowej spada liczba epok potrzebnych sieci do wyuczenia się do tej samej określonej skuteczności, po której osiągnięciu dalsza zmiana wartości skuteczności oraz spadek funkcji kosztu zachodzi bardzo wolno. Jednak dla ostatniego z rozpatrywanych przedziałów [-1.0, 1.0] szybkość uczenia sieci neuronowej zmniejsza się w stosunku do poprzednich mniejszych zakresów. Z poniższych wyników można więc powiedzieć, że najbardziej optymalnym zakresem inicjalizacji wag początkowych jest przedział o bezwzględnej krańcowej wartości mniejszej od 1, ale też nie bardzo bliskiej zeru.

5.2 Eksperyment 3. Zbadanie wpływu różnych metod inicjalizacji wag początkowych na proces uczenia

Założenia:

- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - 30 epok
- zmianom ulegał inicjalizator wag zastosowany w modelu sieci, każdy z nich korzystał z innej metody dobrania wartości wag początkowych

Przebieg eksperymentu:
Wpływ metody inicjalizacji wag początkowych zbadano poprzez wywoływanie procedury uczącej sieć neuronową z zachowanymi domyślnymi parametrami modelu i uczenia, zmieniając tylko zastosowaną regułę losowania wag. Za każdym razem model uczono do uzyskania warunku stopu, czyli do 30 epok. Wyniki zobrazowano na wykresach, które przestawiają zmianę funkcji straty oraz skuteczności otrzymywanej dla każdej metody dla ciągu walidacyjnego. Użyto takich algorytmów inicjalizacji wag, jak:

a) **zero init** - inicjalizacja wszystkich początkowych wag na zero

b) **normal init** - wylosowanie początkowych wag z rozkładu normalnego

c) **random init** - losowa inicjalizacja wag dobrana pod zastosowaną funkcję przejścia, wagi losowane z rozkładu jednorodnego dla przedziału, którego krańce wylicza się według wzoru:

$$[\frac{-a}{\sqrt{n_{in}}}, \frac{a}{\sqrt{n_{in}}}], \tag{1}$$

gdzie n_{in} - liczba połączeń wchodzących do danej warstwy sieci, a - parametr dobierany pod zastosowaną funkcję aktywacji (sigmoida: $a = 2.38$, relu: $a = 2.0$, tanh: $a = 1.0$)

d) **randomization init** - randomizacja wag z rozkładu jednorodnego dla przedziału wyliczanego według reguły:

$$[-\sqrt[n_{in}]{N_h}, \sqrt[n_{in}]{N_h}], \tag{2}$$

gdzie n_{in} - liczba połączeń wchodzących do danej warstwy sieci, N_h - liczba neuronów w warstwie ukrytej

e) **xavier** - inicjalizacja wag metodą Xaviera z rozkładu jednorodnego dla wyliczanego zakresu:

$$[-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}], \tag{3}$$

gdzie fan_{in} - liczba wchodzących połączeń do warstwy, fan_{out} - liczba wychodzących połączeń z warstwy
Metoda została opracowana, aby zapobiec zbieganiu wartości gradientów do zera przy trenowaniu głębszych sieci neuronowych. W założeniu ma ona jak najlepiej zachowywać wariancję wyjść aktywacji i gradientów, dzięki czemu ryzyko ich wyciszenia w trakcie uczenia zmniejsza się.

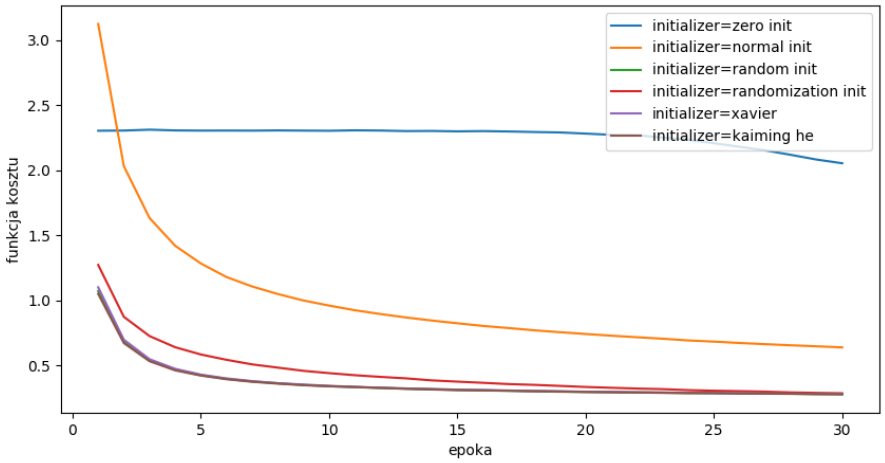
f) **kaiming he** - inicjalizacja wag metodą Kaiming He polegająca na początkowym wylosowaniem wag z rozkładu normalnego, a następnie przeskalowaniu tych wartości przez wyliczaną wartość ze wzoru:

$$scale = \sqrt{\frac{2}{fan_{in}}}, \tag{4}$$

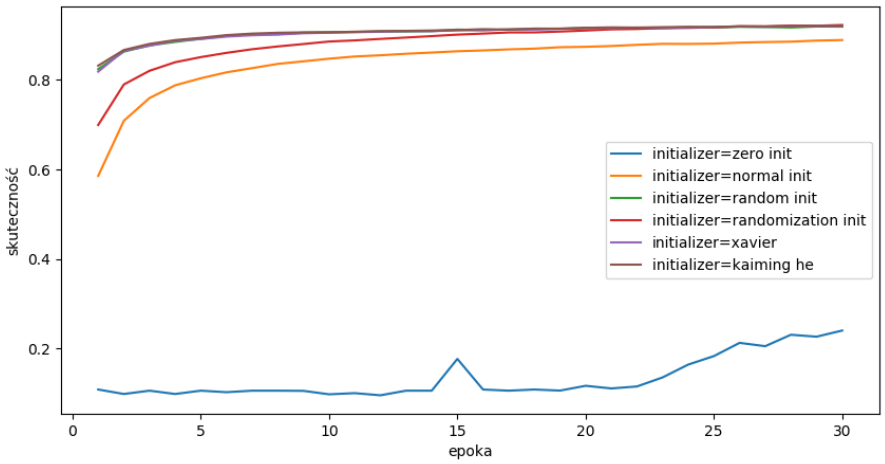
gdzie fan_{in} - liczba wchodzących połączeń do warstwy
Sposób opracowany do trenowania głębszych sieci neuronowych z funkcją przejścia ReLU. Jest ona funckją nieliniową i niesymetryczną, a jej wyjścia mają odchylenie standardowe przesunięte od 1 o pewną skalę. Przeskalowanie zainicjalizowanych wag o te przesunięcie umożliwiło zachowanie rozkładu wyjść aktywacji bliski rozkładowi normalnemu, co okazało się dobrym założeniem do uczenia sieci głębokich, minimalizując ryzyko eksplodowania lub zaniku gradientów.

Przy każdej metodzie wartości początkowe *bias* były inicjalizowane na zero.

Wyniki:



Rysunek 13: Wartość funkcji straty na ciągu walidacyjnym po każdej epoce dla różnych metod inicjalizacji wag



Rysunek 14: Skuteczność na ciągu walidacyjnym po każdej epoce dla różnych metod inicjalizacji wag

Komentarz:
Podobnie jak w **Eksperymencie 2** od razu widać, że nadanie zerowych wartości początkowych wag nie jest dobrą praktyką. Przy takich wagach model nie uczy się, aktualizacja wag zachodzi ciągle o tę samą wartość wyliczanego gradientu wumnażanego przez zero, co można zaobserwować po niskiej wartości skuteczności oraz wysokiej wartości funkcji kosztu na ciągu walidacyjnym po każdej epoce. Przy pozostałych metodach inicjalizacji sieć uczy się, szybko osiągając stabilizację zmian wykresu funkcji straty ciągu walidacyjnego (widoczne wypłaszczenie wykresu po pewnej liczbie epok). Losowanie wag z rozkładu normalnego oraz metoda ich randomizacji osiągają swoje minimum funkcji straty po ok. **15**-tej epoce uczenia, natomiast pozostałe metody (kaiming he, xavier, random init) - po ok. **5**-tej epoce, na wykresie zmiany kosztu linie obrazujące te metody praktycznie pokrywają się. Ostatnie wymienione trzy metody wyróżniają się na tle innych, co widać po wysokiej skuteczności osiągniętej na ciągu walidacyjnym po małej liczbie epok.

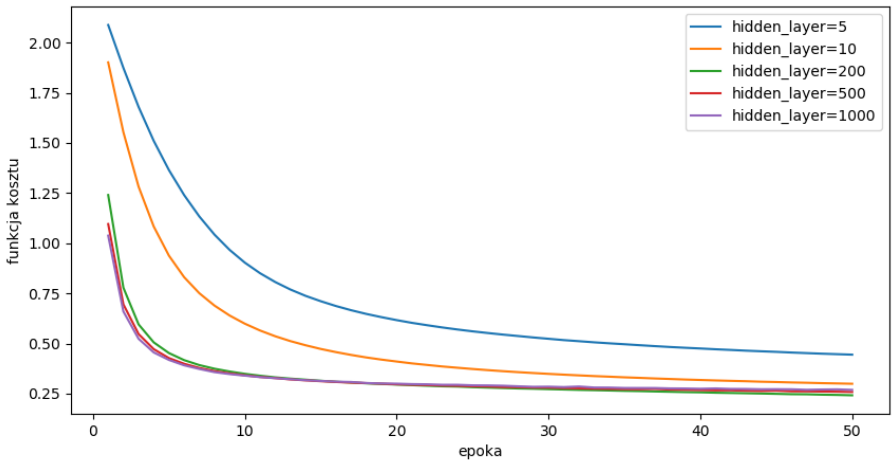
6 Eksperyment 4. Wpływ liczby neuronów w warstwie ukrytej na proces uczenia sieci

Założenia:

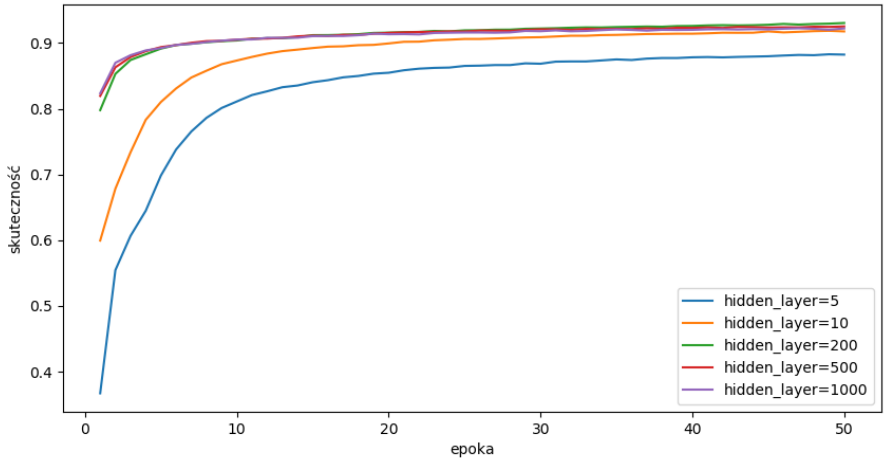
- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - **50** epok
- zmianom ulegała liczba neuronów w warstwie ukrytej modelu sieci MLP, przyjmowały kolejno wartości ze zbioru {5, 10, 200, 500, 2000}

Przebieg eksperymentu:
Eksperyment wykonano poprzez iteracyjne wywoływanie procedury uczącej dla sieci MLP, kolejno zmieniając wartość liczby neuronów w warstwie ukrytej od 5 do 2000 z podanego w założeniach zbioru. Warunkiem stopu dla uczenia było osiągnięcie liczby epok uczenia równej 50. Dla każdej wartości liczby neuronów porównano zmiany wartości skuteczności na ciągu treningowym i walidacyjnym w celu sprawdzenia, czy nie doszło do przeuczenia (*overfitting*) modelu. Zestawiono również na wspólnym wykresie funkcję straty oraz skuteczność na samym ciągu walidacyjnym dla wszystkich zadanych wartości liczby neuronów. Zmierzono czas trwania uczenia w sekundach dla każdej pojedynczej procedury uczącej oraz wyliczono liczbę parametrów całej sieci w poszczególnych zmianach w jej architekturze.

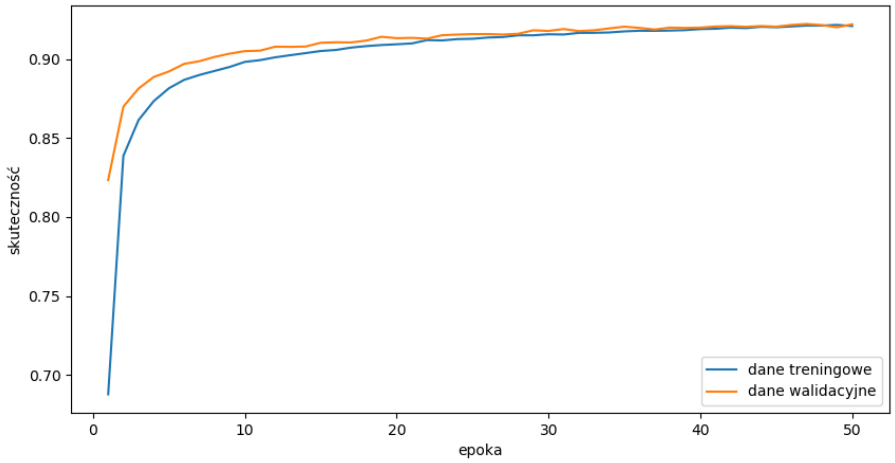
Wyniki:



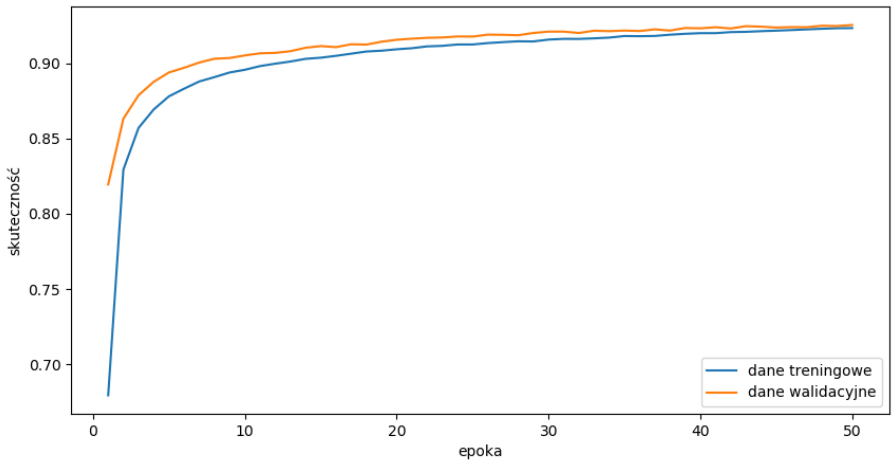
Rysunek 15: Wartość funkcji straty na ciągu walidacyjnym po każdej epoce dla różnych wartości liczby neuronów w warstwie ukrytej



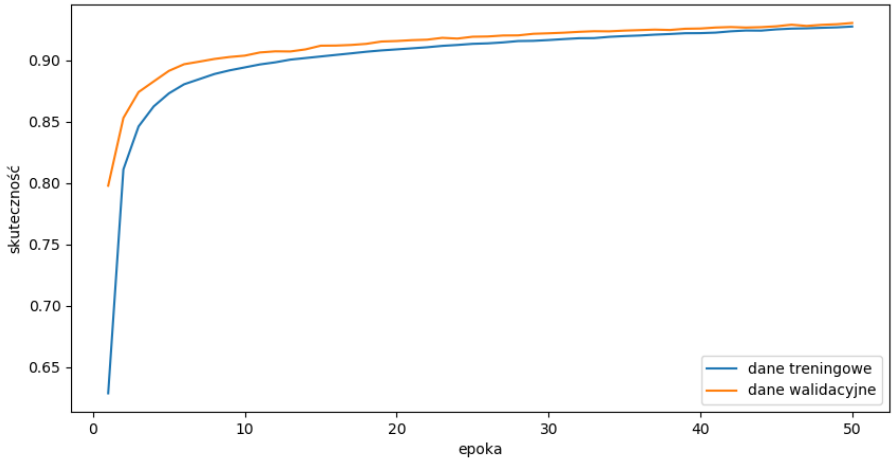
Rysunek 16: Skuteczność na ciągu walidacyjnym po każdej epoce dla różnych wartości liczby neuronów w warstwie ukrytej



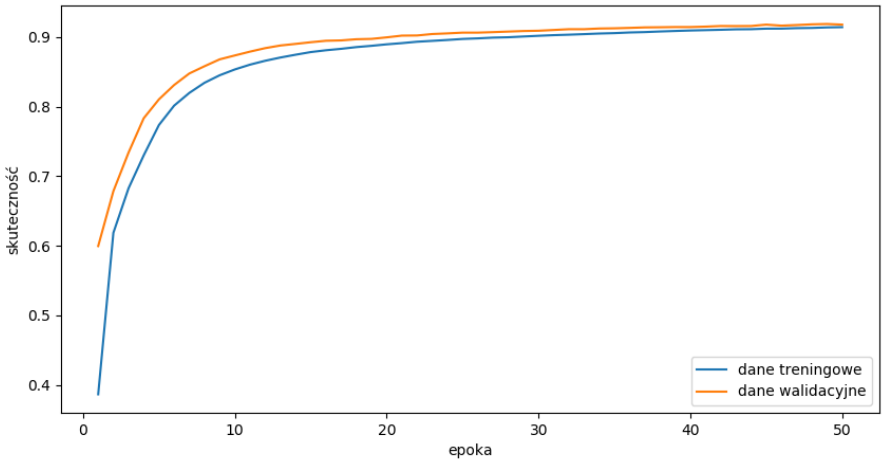
Rysunek 17: Skuteczność na ciągu walidacyjnym i treningowych po każdej epoce dla **1000** neuronów w warstwie ukrytej



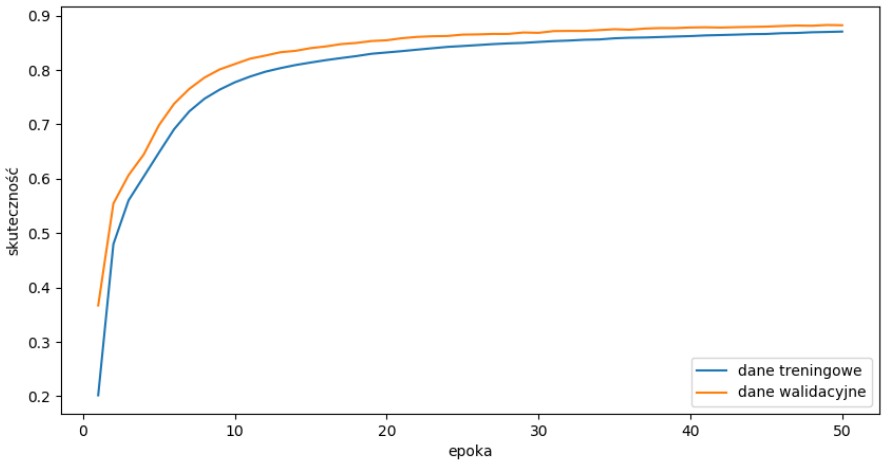
Rysunek 18: Skuteczność na ciągu walidacyjnym i treningowych po każdej epoce dla **500** neuronów w warstwie ukrytej



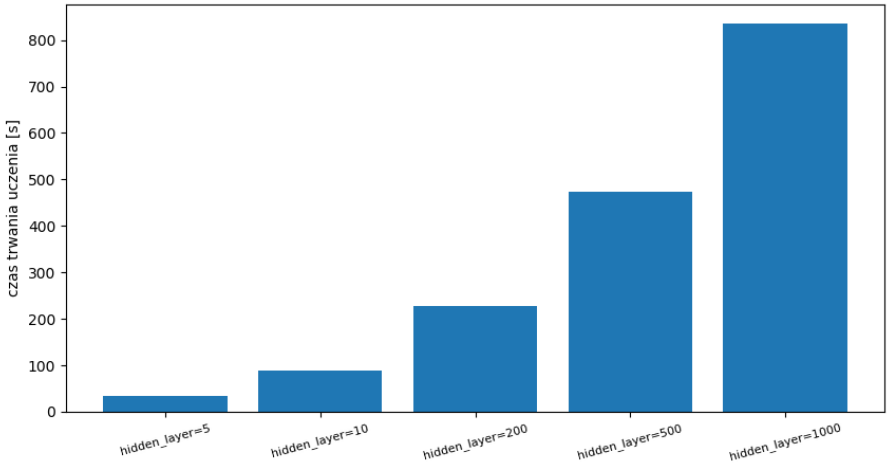
Rysunek 19: Skuteczność na ciągu walidacyjnym i treningowych po każdej epoce dla **200** neuronów w warstwie ukrytej



Rysunek 20: Skuteczność na ciągu walidacyjnym i treningowych po każdej epoce dla **10** neuronów w warstwie ukrytej



Rysunek 21: Skuteczność na ciągu walidacyjnym i treningowych po każdej epoce dla **5** neuronów w warstwie ukrytej



Rysunek 22: Czas uczenia sieci dla różnych wartości liczby neuronów w warstwie ukrytej

Tabela 2: Liczba parametrów w całym modelu sieci dla różnej liczby neuronów warstwy ukrytej

liczba neuronów	5	10	200	500	1000
liczba parametrów	3 985	79 510	159 010	397 510	795 010

Komentarz:
Na **Rysunku 15** i **16** można zaobserwować szybszy spadek funkcji straty wraz ze wzrostem liczby neuronów w warstwie ukrytej architektury sieci, co w konsekwencji skutkuje szybszym wzrostem skuteczności predykcji na ciągu walidacyjnym. Model posiadający więcej neuronów w warstwie ukrytej ma większą łączną liczbę parametrów, co pozwala na wyuczenie się większej liczby cech charakteryzujących dane treningowe. Można więc powiedzieć, że im liczba neuronów jest wyższa, tym sieć szybciej się uczy.
Na rysunkach **17-21** zestawiono porównanie skuteczności predykcji osiąganej przez uczoną sieć dla ciągu treningowego oraz walidacyjnego dla różnych rozmiarów warstwy *hidden*. Niestety, nie zaobserwowano zjawiska przeuczenia sieci przy zwiększającej się liczbie neuronów. Wręcz przeciwnie, uzyskana skuteczność na ciągu walidacyjnym była minimalnie wyższa w każdej próbie eksperymentalnej. Najpewniej jest to związane z użytym do badań zbiorem danych, który jest duży i posiada jednakowy rozkład klas. Wzorce uczące nie są też prawdopodobnie bardzo skomplikowane, dzięki czemu sieć widzi dobrze charakterystyczne dla nich cechy i jest w stanie poprawnie zaklasyfikować dane, nawet przy mniejszym spodziewanym stopniu generalizacji uczenia (jaki może wystąpić w przypadku nadmiaru liczby parametrów modelu). Wzorce w ciągu walidacyjnym najprawdopodobniej musiały być prostszym zbiorem niż treningowy, gdyż skuteczność dla niego była wyższa, co oznacza że wyuczony model mniej razy pomylił się w predykcji klas po 50 epokach. W trakcie trwania eksperymentu próbowano również zmniejszać liczbę wzorców uczących w ciągu treningowym i walidacyjnym, ale nie dało to oczekiwanych rezultatów dobrze widocznego przeuczenia sieci, dlatego zrezygnowano z prezentacji tych wyników. Możliwe także, że uczono model do zbyt małej liczby danych.
Na **Rysunku 22** zaprezentowano zależność czasu trwania uczenia w sekundach od liczby neuronów w warstwie ukrytej. Większa liczba parametrów sieci wymaga dłuższego czasu jej uczenia dla zadanej liczby epok niż w przypadku mniejszych sieci.

7 Eksperyment 5. Zbadanie wpływu wybranych funkcji przejścia na uczenie sieci

Założenia:

- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - 30 epok
- zmianom ulegała funkcja przejścia - wybrano trzy rodzaje funkcji do badań: sigmoidalną, ReLU oraz tangens hiperboliczny.

Przebieg eksperymentu:
Wpływ rodzaju wybranej funkcji aktywacji zbadano, uruchamiając kolejno proces uczenia sieci dla różnych funkcji przejścia do momentu osiągnięcia warunku stopu, którym było 30 epok uczących. Wyniki zestawiono na wspólnych dla wszystkich funkcji aktywacji wykresach, jednym prezentującym zmianę wartości funkcji straty na ciągu walidacyjnym oraz drugim obrazującym osiąganą skuteczność na tym samym ciągu danych. Badane funkcje to:

a) **sigmoidalna** - neliniowa funkcja przyjmująca wartości między 0 a 1

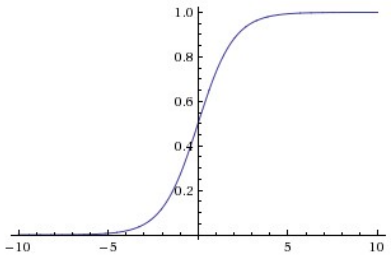
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

b) **tangensa hiperbolicznego** - nieliniowa funkcja przyjmująca wartości od -1 do 1, jej wyjście jest więc skupione wokół zera

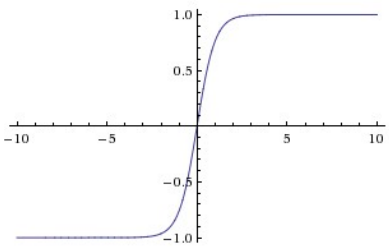
$$tanh(x) = 2\sigma(2x) - 1 \tag{6}$$

c) **ReLU** - jedna z najczęściej stosowanych funkcji przejścia w ostatnim czasie, jej aktywacja jest progiem zerowym - przepuszcza wartości wyłącznie większe od tej wartości

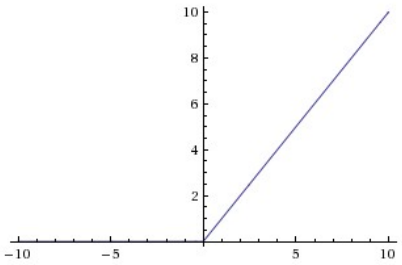
$$f(x) = max(0, x) \tag{7}$$



(a) funkcja sigmoidalna



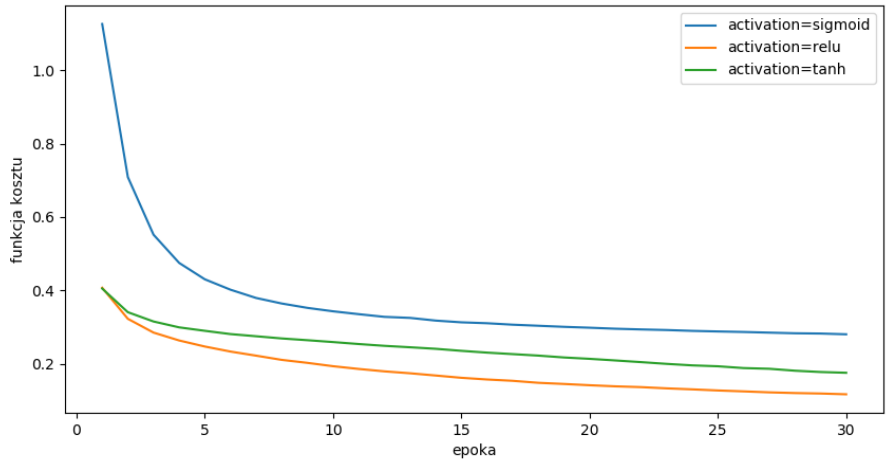
(b) funkcja tangensa hiperbolicznego



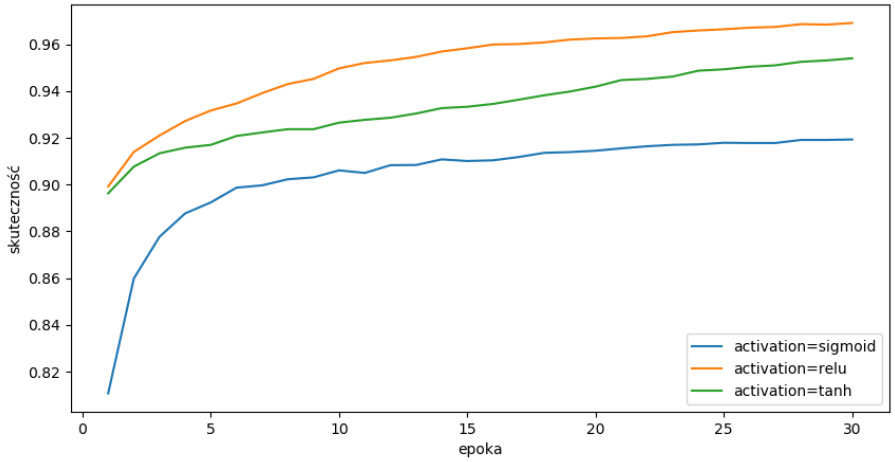
(c) funkcja ReLU

Rysunek 23: Wybrane funkcje przejścia

Wyniki:



Rysunek 24: Wartość funkcji straty na ciągu walidacyjnym dla różnych funkcji przejścia



Rysunek 25: Skuteczność na ciągu walidacyjnym dla różnych funkcji przejścia

Komentarz:
Można zaobserwować, że największy i najszybszy spadek funkcji kosztu osiąga sieć uczona z funkcją przejścia ReLU, dzięki czemu sieć osiągnęła wtedy najwyższą skuteczność w porównaniu z innymi funkcjami aktywacji po tej samej liczbie epok. Następnie najlepiej wypadła funkcja tangensa hiperbolicznego, dla której model osiągnął trochę niższą skuteczność niż dla ReLU. Sieć używająca sigmoidy jako funkcji przejścia wypadła w wynikach najsłabiej, ale uzyskana skuteczność była także wysoka. Spadek funkcji straty dla tej funkcji był najwolniejszy, co szczególnie widać w początkowych epokach uczenia. Sieć uczona z ReLU i tangensem hiperbolicznym po pierwszej epoce uczenia osiąga wartość funkcji kosztu w pobliżu 0.4 (gdzie początkowa jej wartość dla entropii krzyżowej dla dziesięciu klas jest w pobliżu 2.3), podczas gdy wyniki dla sigmoidy przyjmują wtedy wartości jeszcze powyżej 1, czyli ponad dwa razy więcej.
Patrząc na przebieg wykresów na **Rysunku 24** zaobserwowano również, że ich wypłaszczenie następuje w okolicach 10 epoki dla każdej funkcji przejścia, potem tempo uczenia dla każdej z nich stabilizuje się. Różne osiągane skuteczności sieci przy zastosowaniu poszczególnych funkcji aktywacji są więc zależne od innego przebiegu uczenia z daną funkcją w początkowych epokach.
Uzyskane wyniki można wytłumaczyć inną charakterystyką użytych funkcji aktywacji. Dla ReLu

po stronie dodatniej osi wyliczany gradient będzie miał większe wartości niż w przypadku innych badanych funkcji przejścia, co przyśpiesza zbieganie metody gradientu stochastycznego w kierunku minimum. Dzięki temu model uczony z ReLU osiągnął najwyższą skuteczność i uczył się najszybciej. Przebieg funkcji sigmoidalnej posiada łagodniejsze przegięcia na krańcach przedziału dziedziny, w których następuje zbieganie wartości funkcji do wartości 0 lub 1 - pochodna tej funkcji będzie miała więc mniejsze wartości niż w przypadku tangensa hiperbolicznego, dla której przegięcia na tych krańcach są bardziej ostre. Dodatkowo funkcja *tanh* przyjmuje wartości skupione wokół zera przeciwnie do sigmoidy, która wchodzące ujemne wartości zbiera właśnie do 0. Przy tangensie jest więc mniejsze ryzyko zaniknięcia gradientu w procesie uczenia.

8 Eksperyment 6. Wpływ różnych metod optymalizacji współczynnika uczenia

Założenia:

- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - 30 epok
- zmianom ulegał używany optymalizator współczynnika uczenia

Przebieg eksperymentu:
Zbadano wpływ różnych optymalizatorów współczynnika uczenia (***learning rate***) na szybkość uczenia sieci neuronowej poprzez kolejne uruchomienia uczenia modelu do uzyskania warunku stopu 30-stu epok, za każdym razem zmieniając rodzaj zastosowanej metody optymalizującej. Wyniki dotyczące przebiegu funkcji straty oraz skuteczności na ciągu walidacyjnym danych dla każdej z nich zestawiono na wspólnych wykresach. W ćwiczeniu zbadano następujące optymalizatory:

a) **SGD** (*Stochastic Gradient Descent* - zmiana wag następuje w przeciwnym kierunku proporcjonalnie do gradientu błędu przemnożonego przez współczynnik uczenia η (stały hiperparametr). W trakcie eksperymentów zbadano wartość $\eta = 0.1$ oraz $\eta = 0.01$

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{8}$$

gdzie θ - uczone wagi, η - współczynnik uczenia, $\nabla_\theta J(\theta)$ - gradient błędu

b) **SGD momentum** - uwzględnia zmianę wag z poprzedniego kroku, wprowadzając dodatkową zmienną przyśpieszenia (***velocity***) oraz stały hiperarametr współczynnika momentum, pomaga przeskoczyć płaskie miejsca funkcji (*plateu*) w celu znalezienia optymalnego minimum. Hiperparametry ustawiono następująco: $\gamma = 0.9$, $\eta = 0.01$

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{9}$$

$$\theta = \theta - v_t \tag{10}$$

gdzie v_t - przyśpieszenie w aktualnym kroku, v_{t-1} - przyśpieszenie z poprzedniego kroku, η - współczynnik uczenia, $\nabla_\theta J(\theta)$ - gradient błędu, γ - współczynnik momentum

c) **NAG** (*Nesterov Accelerated Gradient*) - momentum Nesterova, zapewnia silniejszą konwergencję dla funkcji wypukłych i w praktyce działa nieco lepiej niż zwykłe momentum. Zakłada "patrzenie wprzód" poprzez wyliczanie gradientu błędu względem przyszłej pozycji wektora parametrów (a nie jego aktualnej pozycji), co zapobiega zbyt dużemu przyśpieszeniu i ryzyku przeskoczenia minimum. Hiperparametry γ oraz η ustawiono na takie same wartości jak dla zwykłego momentum.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{11}$$

$$\theta = \theta - v_t \tag{12}$$

W praktyce obliczanie momentum Nesterova wykonuje się na podstawie przekształconych wzorów **11** i **12**:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{13}$$

$$\theta = \theta - \gamma v_{t-1} + (1 + \gamma) v_t \tag{14}$$

gdzie v_t - przyśpieszenie w aktualnym kroku, v_{t-1} - przyśpieszenie z poprzedniego kroku, η - współczynnik uczenia, $\nabla_\theta J(\theta)$ - gradient błędu, $\nabla_\theta J(\theta - \gamma v_{t-1})$ - gradient "wprzód" γ - współczynnik momentum

d) **Adagrad** - metoda adaptująca współczynnik uczenia w zależności od historii zmian wag. Jeśli dotychczasowe zmiany wag zachodziły rzadko, współczynnik będzie większy, a dla częstszych zmian - mniejszy.

$$\theta_{t+1} = \theta_t - \eta_w g_t \tag{15}$$

$$\eta_w = \frac{\eta}{\sqrt{G_t + \epsilon}} g_t \tag{16}$$

$$G_t = \sum (g_t)^2 \tag{17}$$

gdzie η - stała uczenia, domyślnie równa 0.001, g_t - gradient w aktualnym kroku, G_t - suma kwadratów gradientów do aktualnego kroku, η_w - zmienny współczynnik uczenia

e) **Adadelta** - rozszerzenie Adagradu mające zapobiec zbyt dużemu spadkowi wartości współczynnika uczenia. Nie kumuluje historii wszystkich minionych gradientów, ale sumuje ich średnią kwadratową w pewnym przesuwającym się oknie przeszłych iteracji uczenia. W oryginalnej wersji nie wymaga podawania współczynnika uczenia. Stałe wymagane do wzorów przyjęto: $\gamma = 0.95$, $\epsilon = 10^{-6}$.

$$\Delta\theta_t = -\frac{RMS[\delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{18}$$

$$RMS[g]_t = \gamma E[g]^2_{t-1} + (1-\gamma) g_t^2 \tag{19}$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{20}$$

$$\tag{21}$$

gdzie $RMS[g]_t$ (*root mean square*) - akumulator gradientów, g_t - gradient w aktualnym kroku, $\Delta\theta_t$ - wartość aktualizacji wag, $RMS[\Delta\theta]_t$ - akumulator zmian wag

f) **RMSprop** - nieopublikowana oficjalnie metoda, która także nie wymaga podania współczynnika uczenia. Wzór ma podobną postać i oznaczenia do Adadelty:

$$E[g^2]_t = 0.9 E[g^2]_{t-1} + 0.1 g_t^2 \tag{22}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \tag{23}$$

Przyjęto $\eta = 0.001$ oraz $\epsilon = 10^{-6}$.

g) **Adam** (*Adaptive Moment Estimator*) - metoda optymalizacji podobna do RMSprop, ale z uwzględnieniem momentum. Jedna z częściej stosowanych aktualnie metod optymalizacji przy uczeniu sieci neuronowych. Stałe użyte we wzorach przyjmują wartości: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

$$m_t = \beta_1 m_{_{t-1} + (1-\beta_1) g_t} \tag{24}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g^2 \tag{25}$$

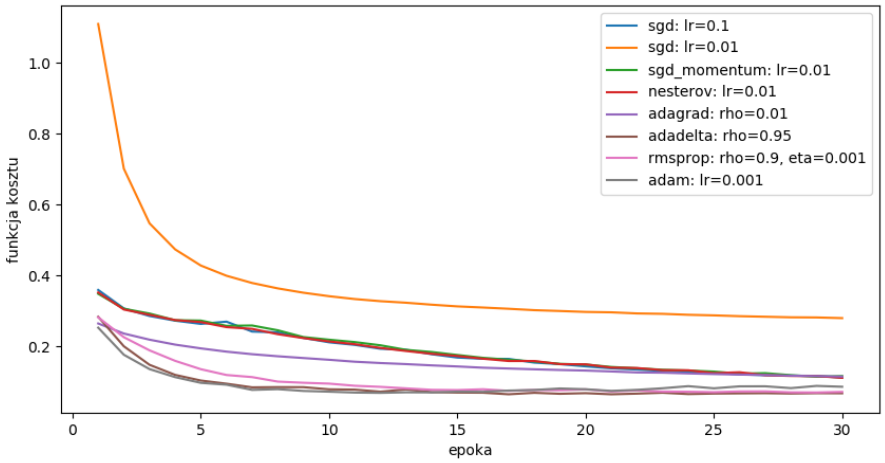
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{26}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{27}$$

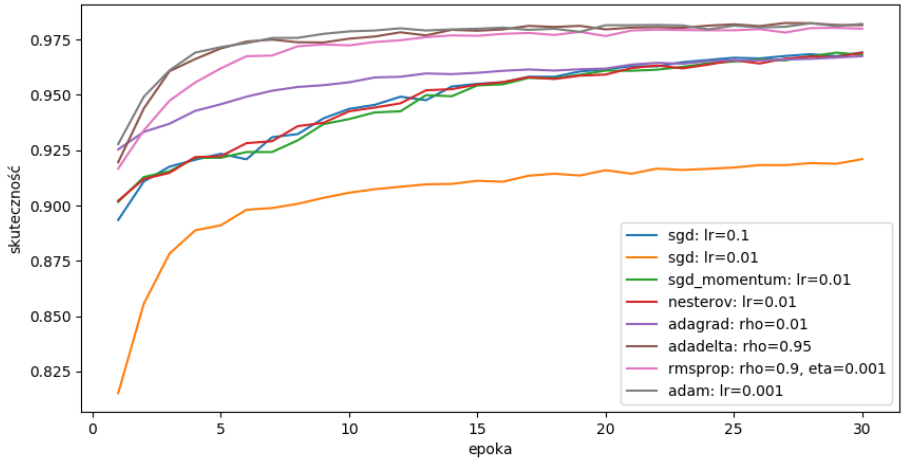
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \tag{28}$$

gdzie m_t - estymator pierwszego momentu (średniej) gradientów, v_t - estymator drugiego momentum (wariancji) gradientów, θ_{t+1} - wyliczone wartości wag dla kolejnego kroku uczenia, θ_t - wagi aktualnego kroku uczenia

Wyniki:



Rysunek 26: Wartość funkcji straty na ciągu walidacyjnym liczona po każdej epoce dla różnych optymalizatorów współczynnika uczenia



Rysunek 27: Skuteczność na ciągu walidacyjnym liczona po każdej epoce dla różnych optymalizatorów współczynnika uczenia

Tabela 3: Skuteczność na ciągu **testowym** uzyskana dla różnych optymalizatorów po 30 epokach uczenia

ciąg testowy	
optymalizator	skuteczność
SGD ($\eta = 0.1$)	96.69%
SGD ($\eta = 0.01$)	91.76%
SGD momentum	96.64%
NAG	96.57%
Adagrad	96.67%
Adadelta	98.13 %
RMSprop	98.15 %
Adam	98.32 %

Komentarz:
Z analizy wyników na **Rysunku 27** można zaobserwować, że najwyższą skuteczność na ciągu walidacyjnym uzyskał model trenowany z użyciem optymalizatorów **Adam**, **Adadelta** oraz **RMSprop** (wszystkie ponad 97.5%). Są to bardzo podobne algorytmy polegające na adaptacyjnym dobieraniu współczynnika uczenia do zachodzących w ciągu uczenia zmian gradientów oraz dynamiki aktualizacji wag. Najwyższą skuteczność na ciągu testowym odnotowano dla Adama, który minimalnie przewyższył wyniki dla poprzednich dwóch optymalizatorów. Należy jednak zaznaczyć, że eksperyment wykonana tylko jeden raz i wymagałoby to większej liczby powtórzeń.
Kolejną grupą optymalizatorów, które uzyskały podobną do siebie skuteczność na poziomie ponad 95% to **Adagrad**, **NAG**, **SGD z momentum** oraz zwykłe **SGD** z większym współczynnikiem uczenia równym **0.1**. Wyróżnia się z nich metoda Adagrad, która w początkowych epokach uczenia znacznie przewyższała uzyskaną skutecznością i w tempie spadku funkcji straty pozostałe porównywane z nią optymalizatory. Po 15 epoce uczenia jednak wykres jej skuteczności wyrównał się z pozostałymi. Jest to spowodowane najpewniej tym, że im dłuższy czas uczenia, tym współczynnik η staje się w optymalizacji Adagrad coraz to mniejszy, co nie pozwoliło na osiągnięcie wyższej skuteczności i spowolniło tempo spadku funkcji kosztu.
Najmniejszą skuteczność uzyskał zwykły **SGD z** $\eta = 0.01$, co było przewidywalne, gdyż pozostałe optymalizatory miały na celu kolejno ulepszać metodę gradientu prostego. Ustawienie jednak hiperparametru na 0.1 zwiększyło wartość aktualizacji wag w pojedynczym kroku, co przyśpieszyło spadek funkcji kosztu i sieć osiągnęła w tym przypadku skuteczność na poziomie Adagrad.

9 Eksperyment 7. Wpływ funkcji straty na szybkość uczenia sieci

Założenia:

- stałe domyślne parametry modelu sieci oraz procedury uczącej wymienione w **pkt. 1**
- warunek stopu - **60** epok
- zmianom ulegała funkcja kosztu - użyto entropii krzyżowej (*cross entropy*) poprzedzonej softmaxem oraz błędu średniokwadratowego poprzedzonego funkcją tożsamościową

Przebieg eksperymentu:
Zbadano wpływ dwóch różnych funkcji straty na proces uczenia się sieci MLP poprzez uruchomienie procedury uczącej dla każdej z nich aż do uzyskania warunku stopu 60 epok uczenia. Sprawdzono ich przebieg na ciągu treningowym w celu zobrazowania tempa spadku oraz zestawiono na wspólnym wykresie wyniki skuteczności modelu po każdej epoce dla każdej z wybranych funkcji.

a) **entropia krzyżowa** (*cross entropy*) - wskazuje odległość pomiędzy predykcją sieci a prawdziwą wartością. Jest używana, gdy wyjście sieci może być rozumiane jako reprezentujące prawdopodobieństwo, że każda hipoteza może być prawdziwa, tzn. gdy jest rozkładem prawdopodobieństwa. Z tego powodu w sieciach trenowanych z zastosowaniem tej funkcji straty stosuje się sigmoidę lub softmax jako aktywację ostatniej warstwy wyjściowej.

$$H(y,p) = -\sum_i y_i log(p_i) \tag{29}$$

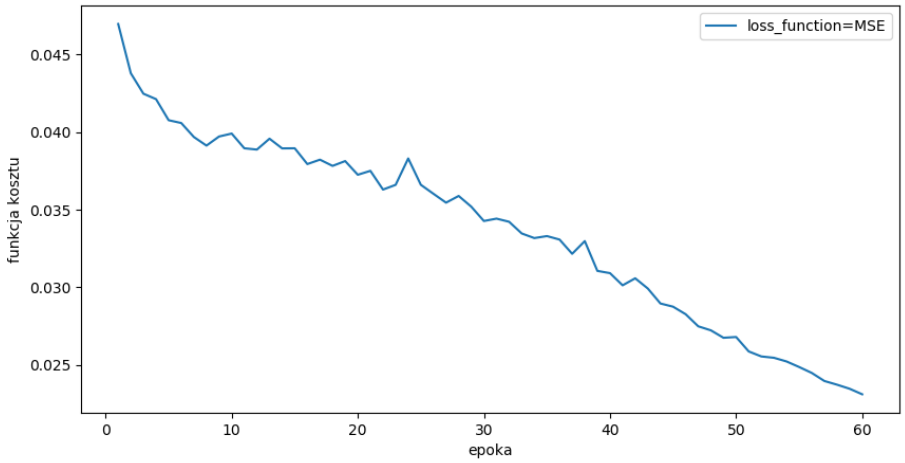
gdzie p_i - prawdopodobieństwo danej etykiety z predykcji, y_i - prawdziwa etykieta

b) **błąd średniokwadratowy MSE** (*mean squared error* - mierzy średnią kwadratów błędów, czyli średnią różnicę między predykcją a prawdziwą wartością.

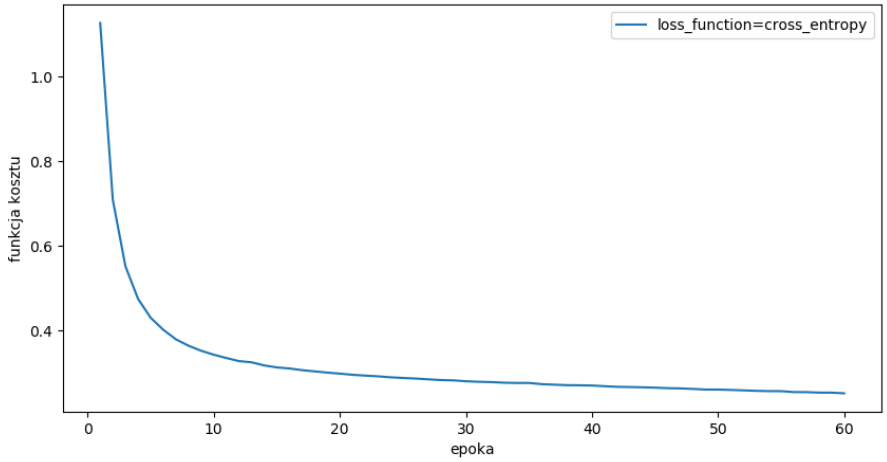
$$MSE = \frac{1}{n}\sum(y_i - \hat{y}_i)^2 \tag{30}$$

gdzie y_i - prawdziwa etykieta, $\hat{y}_i$ - predykcja etykiety dla danego wzorca uczącego

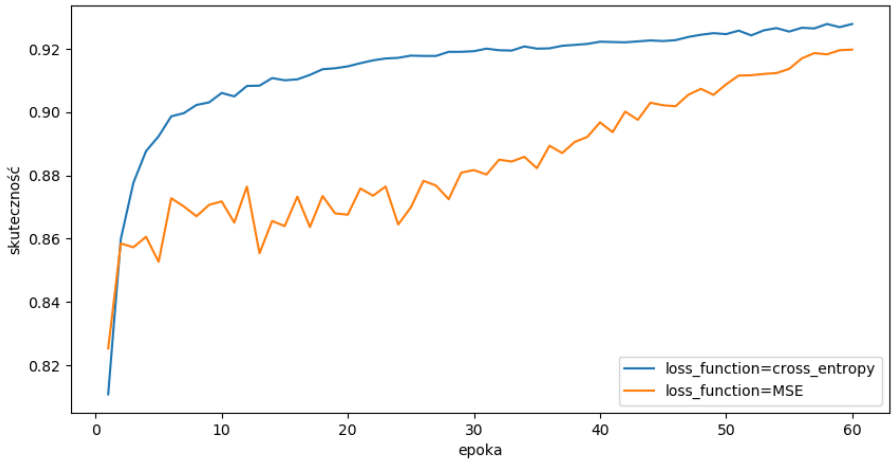
Wyniki:



Rysunek 28: Wartość funkcji kosztu na ciągu walidacyjnym liczona po każdej epoce dla funkcji błędu średniokwadratowego



Rysunek 29: Wartość funkcji kosztu na ciągu walidacyjnym liczona po każdej epoce dla entropii krzyżowej



Rysunek 30: Skuteczność na ciągu walidacyjnym liczona po każdej epoce dla różnych funkcji straty

Komentarz:
Na podstawie wyników można stwierdzić, że entropia krzyżowa znacznie szybciej spada w początkowych epokach uczenia, następnie tempo jej spadku stabilizuje się - ma przebieg hiperboliczny. Funkcja MSE zmniejsza swoją wartość stopniowo z każdą epoką, cechuje się małymi skokami wzrostowymi i spadkowymi. Analizując wyniki na **Rysunku 30** widać, że wyższą skuteczność osiągnęła sieć uczona z entropią krzyżową. Wykres skuteczności dla *cross entropy* rósł gwałtownie do ok. 10 epoki, następnie tempo jej wzrostu utrzymywało się na stałym poziomie. Skuteczność sieci z MSE wypadła znacznie gorzej, dopiero ok. 60 epoki udało się uzyskać skuteczność przewyższającą 90%.
Entropia krzyżowa jest więc znacznie lepszą miarą dla rozwiązywanego problemu klasyfikacji niż MSE, ponieważ granica decyzji w zadaniu klasyfikacyjnym jest duża w porównaniu np. z regresją, dla której błąd średniokwadratowy jest najczęściej stosowany. Entropia krzyżowa znacznie bardziej karze błędne klasyfikacje modelu niż błąd średniokwadratowy, który bada odległość predykcji od poprawnego wyniku.

10 Podsumowanie wyników badań eksperymentalnych

1. **Wielkość paczki danych treningowych:**
   Podział danych na paczki (metoda gradientu stochastycznego) jest znalezieniem kompromisu między uczeniem po pojedynczym wzorcu i po całym zbiorze treningowym. Aktualizacja wag po całym ciągu treningowym (metoda gradientu prostego) powoduje, że wyliczany gradient po epoce uczenia zmierza kierunkowo do pewnego jednego minimum lokalnego. Nie jest to najlepsze, gdyż może pominąć inne minima, które mogły okazać się korzystniejszymi i zapewnić wyższą skuteczność. Dodatkowo, ucząc po całym zbiorze danych wymagana jest większa liczba epok do uzyskania satysfakcjonującej skuteczności wyuczenia sieci. Z tego względu praktykuje się wybór relatywnie małej wielkości batcha - często blisko wartości 32. Uczenie mniejszymi paczkami wprowadza pewien szum w trakcie uczenia, co daje efekt regularyzacji oraz zmniejsza błąd generalizacji sieci, daje też większe prawdopodobieństwo wyskoczenia z lokalnego minimum i być może znalezienie lepszego. Mniejsza paczka umożliwia także łatwiejsze przetwarzanie danych w pamięci podręcznej komputera, jednak zbyt mała wielkość batcha może znacząco wydłużyć procedurę uczenia, gdyż ciągłe aktualizacje wag wymagają dłuższego czasu.
2. **Metoda inicjalizacji wag początkowych:**
   - Inicjalizowanie wartości wag początkowych na zero zaburza całkowicie proces uczenia sieci. Wartości wychodzące z warstw przyjmują te same wartości - przemnożone przez zerowe wagi dadzą wyjście z sieci równe zero. W konsekwencji gradienty wyliczane podczas propagacji wstecznej zanikają, przez co aktualizacja wag nie zachodzi.
   - Przedział, z którego będą losowane wagi najlepiej dobrać symetrycznie, wybierając bezwzględną wartość jego krańca na trochę większą od zera, ale nie większą niż 1.
   - Istnieje wiele metod służących do optymalizacji procesu losowania wag początkowych w sieci neuronowej. Niewątpliwie wyróżniającymi się na tle innych jest metoda Kaiming He oraz Xavier. Podobnie do nich wypadł także sposób losowej inicjalizacji wag dostosowywany pod konkretną użytą w architekturze funkcję przejścia.
3. **Liczba neuronów w warstwie ukrytej:**
   - Model posiadający warstwę ukrytą o większym rozmiarze posiada więcej parametrów uczących, więc sieć jest w stanie nauczyć się większej liczby cech charakteryzujących dany zestaw danych. Dzięki temu proces uczenia przebiega szybciej i sieć osiąga wyższą skuteczność na ciągu walidacyjnym po mniejszej liczbie epok.
   - Zbyt duża liczba neuronów w warstwie ukrytej sieci może prowadzić do ***overfittingu***, czyli stanu przeuczenia sieci, gdy skuteczność dla ciągu treningowego cały czas rośnie, natomiast dla walidacyjnego zaczyna spadać. Sieć ze zbyt dużą liczbą parametrów ma mniejsze zdolności generalizacji - zaczyna zapamiętywać głównie widziane podczas treningu wzorce uczące.
   - Większa liczba neuronów warstwy ukrytej wydłuża potrzebny czas na uczenie sieci dla zadanej takiej samej liczby epok w porównaniu z modelami sieci posiadającymi mniejszą liczbę parametrów.
   - Obserwowanie zjawiska przeuczenia sieci przy zbyt dużej liczbie parametrów uczących jest też zależne od charakterystyki zbioru użytych danych.
4. **Funkcje przejścia:**
   - Funkcja aktywacji ReLU wyróżnia się na tle innych szybkim tempem spadku funkcji straty i osiąganą wysoką skutecznością predykcji.
   - W trakcie uczenia sieci może wystąpić ryzyko zanikania oraz nasycenia gradientów. Jest to dużym problemem przy uczeniu głębokich sieci neuronowych. W tym ćwiczeniu jednak te zjawisko nie powinno wystąpić ze względu na płytką architekturę sieci MLP oraz zastosowaniu metody Xaviera do inicjalizacji wag początkowych.
5. **Optymalizacja współczynnika uczenia:**
   - Dobór optymalizatora zależy od typu problemu i zastosowanej funkcji straty. W ostatnim czasie jednak dość mocno widać popularność metody Adama do procesu uczenia sieci neuronowych, który przewyższa swoją skutecznością pozostałe metody i znajduje zastosowanie dla wielu problemów.

- Spośród badanych optymalizatorów wyróżniały się także RMSprop oraz Adadelta, które pozwoliły uzyskać wysoką skuteczność na ciągu walidacyjnym oraz dodatkowo nie wymagały podania współczynnika uczenia η, gdzie do pozostałych metod podanie tego hiperparametru jest konieczne.

6. **Funkcja straty:**
   - Dobór funkcji kosztu zależy ściśle od rodzaju rozwiązywanego problemu.
   - Do problemu klasyfikacji dobrym wyborem funkcji kosztu jest entropia krzyżowa, ponieważ maksymalizuje ona prawdopodobieństwo poprawnej predykcji sieci. Należy pamiętać wtedy o użyciu funkcji *softmax* jako aktywacji ostatniej warstwy sieci, żeby jej wyjście sprowadzić do rozkładu prawdopodobieństwa.
   - Błąd średniokwadratowy znajduje dobre zastosowanie w przypadku problemów regresji, gdzie wyjście z sieci jest liniowe. W przypadku zadań klasyfikacji nie daje już tak dobrych wyników jak *cross entropy*, gdyż nie karze wystarczająco dobrze błędnych predykcji. Należy też pamiętać, że funkcja straty MSE nie współpracuje dobrze z nieliniowym wyjściem sieci (takim jak softmax bądź sigmoida).