Metody głębokiego uczenia
Sprawozdanie z pierwszego projektu – Zaimplementowanie sieci MLP

Jadwiga Słowik, Paulina Tomaszewska

25 marca 2019

1 Cel zadania

Celem zadania było stworzenie sieci neuronowej, która klasyfikuje obrazki z ręcznie napisaną cyfrą do kategorii odpowiadającej widocznej cyfrze. Jako zbiór danych posłużył zbiór *MNIST*, który składa się z 60000 obrazków treningowych i 10000 testowych. Rozmiar obrazków to 28×28. Występują one jedynie w skali szarości, tzn. są zapisane w postaci jednej macierzy, a nie trzech jak w przypadku obrazków kolorowych (zapisanych w ramach konwencji RGB).

2 Opis rozwiązania

W ramach projektu napisano w języku *Python* (przy pomocy narzędzia *jupyter-notebook*) sieć neuronową jednokierunkową – *MLP*. Przy konstruowaniu sieci skorzystano jedynie z pakietów do obliczeń macierzowych oraz operacji na ramkach danych (nie wykorzystano frameworków dedykowanym dziedzinie *Deep Learning*): *numpy* i *pandas*.

Rozwiązanie zadania składało się z następujących etapów:

1. Wczytanie danych
2. Przetworzenie danych (*ang. preprocessing*)
   Dane zostały przetworzone w celu zamiany wartości pikseli. Początkowo, wartości pikseli były liczbami naturalnymi z przedziału $[0, 255]$. Docelowo, wartości pikseli zostały zmianione tak, by były liczbami rzeczywistymi z przedziału $[0, 1]$ – normalizacja.
3. Implementacja sieci
   Proces uczenia sieci neuronowej składa się z następujących etapów:
   (a) Inicjalizacja wag sieci losowymi wartościami
   (b) Podział danych na zbiór treningowy i walidacyjny
   (c) „Przetasowanie" elementów zbioru treningowego
   (d) Algorytm *Feed-forward* dla zbioru treningowego wraz z obliczeniem funkcji kosztu
   Iteracyjnie, dla kolejnych warstw sieci, obliczany jest wektor wartości wyjściowych (pojedynczy element wektora odpowiada wartości dla odpowiedniego neuronu z danej warstwy) korzystając z następującego wzoru:

$$y = f\_activation_i(W_i \cdot x + b_i) \tag{1}$$

   W_i to macierz wag sieci neronowej (dla i-tej warstwy), x to wektor wejść (wyniki dla neuronów z poprzedniej warstwy), b_i – wektor z wartościami *bias* dla neuronów danej warstwy.

W naszym rozwiązaniu zaimplementowane zostały następujące funkcje aktywacji:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

$$ReLU(x) = max(0,\, x) \tag{4}$$

Ostatnia warstwa miała zawsze ustawioną funkcję aktywacji:

$$softmax(x) = \frac{e^x}{\sum_{i=0}^{n-1} e^{x_i}}, \quad \text{gdzie } n \text{ to długość wektora x} \tag{5}$$

Ponadto, w naszym rozwiązaniu zostały wykorzystane następujące funkcje kosztu (jako p (wektor „predykcji") oznaczamy wektor z wartościami prawdopodobieństw wystąpienia odpowiedniej klasy, natomiast jako l oznaczamy klasę prawdziwą (oczekiwaną) przedstawioną w formie *one-hot-encoded*):

$$MSE(p,\, l) = \frac{\sum_{i=0}^{n-1}(p_i - l_i)^2}{n} \tag{6}$$

$$cross\_entropy(p,\, l) = -\sum_{i=0}^{n-1} l_i \cdot log(p_i + \epsilon) \tag{7}$$

Dodanie ϵ do wartości parametru logarytmu ma na celu wyeliminowanie przypadku próby obliczenia logarytmu z 0. Jako ϵ wybraliśmy wartość e^{-12}.

(e) Propagacja wsteczna przy użyciu algorytmu *SGD* (*ang. Stochastic Gradient Descent*)
Propagacja wsteczna jest wykonywana w celu zaktualizowania wag sieci (oraz wartości *bias*) w taki sposób, aby uzyskać jak najmniejszą wartość funkcji kosztu. Dlatego też, obliczany jest w tym miejscu gradient funkcji kosztu, dzięki któremu jesteśmy w stanie określić kierunek najszybszego spadku wartości funkcji.
Skorzystałyśmy z następujących wzorów ([5]):

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{8}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{9}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{10}$$

$$\frac{\partial C}{\partial w^l_{j,\,k}} = a^{l-1}_k \delta^l_j \tag{11}$$

Wartości a oznaczają kombinację liniową wag i wejść do neuronu powiększone o *bias*.
Używając powyższych wzorów, jesteśmy w stanie zaktualizować wartości wag (i *biasów*) w sieci zgodnie z następującymi wzorami:

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T \tag{12}$$

$$b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}, \quad \text{gdzie } \eta \text{ to stała uczenia, } m \text{ – rozmiar } batcha \tag{13}$$

(f) Algorytm *Feed-forward* (*inference*) dla zbioru walidacyjnego wraz z obliczeniem funkcji kosztu

Punkty *(c) - (f)* powtarzamy tyle razy, ile jest epok.

Dokładniejszy opis zawarty jest w punkcie dotyczącym *ulepszeń podstawowego rozwiązania*.

4. Testowanie przy użyciu wybranych hiperparametrów
   Proces predykcji klasy polega na uruchomieniu funkcji *Feed-forward* (*inference*) dla wytrenowanej sieci.

   Dokładniejszy opis procesu i wyników testowania został zawarty w rozdziale *Wyniki*.

2.1 Architektura rozwiązania

Sieć składa się z kilku warstw. Pierwszą stanowi warstwa wejściowa, która wczytuje obrazki w postaci wektorów 784-elementowych (każdy obrazek ze zbioru ma wymiary 28×28, co daje łącznie 784 piksele). Każdy element wektora jest liczbą rzeczywistą należącą do przedziału $[0, 1]$. Następnie są warstwy ukryte - gęste (*ang. fully-connected*). Użytkownik może sam zdefiniować dla każdej warstwy ukrytej liczbę neuronów a także jaką funkcję aktywacji chce użyć. Warstwa wyjściowa składa się z 10 neuronów - każdy odpowiada jednej z klas (tzn. odpowiedniej cyfrze). W przypadku tej warstwy została użyta funkcja *softmax*, która sprowadza wartości do liczb z przedziału $[0, 1]$, które sumują się do wartości 1. W ten sposób modelujemy prawdopodobieństwo wystąpienia każdej z klas.

2.2 Ulepszenia podstawowego rozwiązania

W rozwiązaniu zastosowano następujące mechanizmy:

1. Zbiór walidacyjny
   Wyodrębniono ze zbioru treningowego zbiór walidacyjny (10000 elementów), który w oparciu o metryki pozwalał na lepsze analizowanie procesu uczenia się sieci. Dzięki temu rozwiązaniu możliwe jest zastosowanie np. *kryterium stopu*, które w sytuacji gdy sieć zaczyna się przeuczać (błąd na zbiorze walidacyjnym zaczyna gwałtownie rosnąć) zatrzymuje trening sieci i wraca do stanu wag sprzed wzrostu wartości błędów.

2. Losowe tasowanie danych (*shuffle*)
   Dane treningowe w każdej epoce były poddawane losowemu przetasowaniu, tak aby *batche* w każdej z iteracji składały się z możliwie różnych rekordów, co zwiększa zdolność sieci do generalizowania i nieprzywiązywania się do kolejności wejściowych danych.

3. Inicjalizacja wag
   W przypadku nieprawidłowej inicjalizacji wag algorytm uczenia może nie zbiegać do rozwiązania, albo robić to wolniej niż w przypadku optymalnego doboru wartości początkowych. Inicjalizacja wag w inny sposób niż poprzez zastosowanie wszędzie wartości zero nazywana jest *złamaniem symetrii*. Generalnie jeśli startowe wartości wag są za małe/duże może pojawić się problem *vanishing/ exploding gradient*. Aby tego uniknąć a także przyspieszyć proces treningu sieci (znalezienia optimum) inicjalizuje się wartości wag z rozkładu $N(0, 1/n_{in})$). Inną często stosowaną metodą jest inicjalizacja Xaviera, zaproponowana przez Glorot & Bengio [4]. Tą też metodę zaimplementowano w naszej sieci neuronowej typu MLP. Wyróżnią się także metodę zwaną MSRA, która jest sugerowana w przypadku użycia w sieci funkcji aktywacji reLU.

4. Optymalizator
   W rozwiązaniu zastosowano algorytm *wstecznej propagacji* zwany *Stochastic Gradient Descent (SGD)* z uwzględnieniem podziału danych treningowych na *batche*. W ten sposób

nie tylko proces uczenia przebiega efektywniej pod względem wartości metryk, ale też możliwe są obliczenia macierzowe, które pozwalają skrócić czas treningu sieci.

Batch
Zazwyczaj stosowanymi rozmiarami *batchy* są: 32, 64, 128, 256 czy 512. W naszym rozwiązaniu zdecydowałyśmy się na 128. Liczebność zbioru treningowego nie stanowiła wielokrotności wielkości rozmiaru *batcha*, wówczas zdecydowano się, nie odrzucać pozostałych rekordów, tylko tworzyć mniej liczny *batch*. Z uwagi na fakt, że zaakumulowane wartości gradientów wag były dzielone przez liczebność batcha, fakt, że jeden z nich był mniejszy nie miał dużego znaczenia na zbieżność algorytmu.

Regularyzacja wag
W celu uniknięcia sytuacji, gdy wartości wag będą bardzo duże, zastosowano *regularyzację* (nie dotyczy to wag typu *bias*). Wówczas, eliminuje się sytuację, gdy mała różnica wartości na wejściu powoduje duże różnice na wyjściu sieci, co sprawia, że sieć traci zdolność generalizacji.

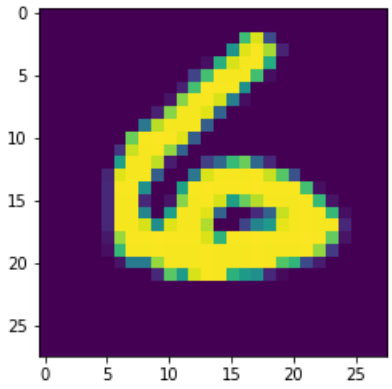
Momentum
W algorytmie dano użytkownikowi możliwość treningu sieci z uwzględnieniem *momentum*. Dzięki jego zastosowaniu trening jest bardziej stabilny (mniej „zygzakowaty"), ponieważ sieć zapamiętuje i bierze pod uwagę także kierunek zmian wag (gradient wag) z poprzedniej iteracji.

5. Data augmentation
   Technika *data augmentation* jest często stosowana ze względu na dwa aspekty:

   - Zauważono, że sieci neuronowe wytrenowane na małych zbiorach danych mają tendencję do przeuczenia (*overfitting*). [2]
   - Oczekuje się, aby algorytmy klasyfikacyjne były w stanie wychwycić kluczowe cechy obrazków niezależnie od skali, rozmiaru, lokalizacji, czy kolorów obiektów na obrazkach (generalizacja).

   W ramach *data augmentation* pewien reprezentatywny podzbiór danych treningowych (o jednostajnym rozkładzie) poddawany jest transformacjom. Stosuje się dwie klasy transformacji: geometryczne i fotometryczne.

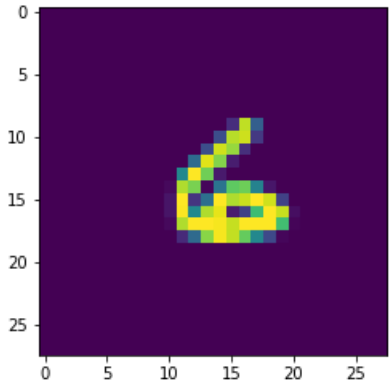
   W ramach pierwszej grupy wyróżnia się *horizontal flipping*, rotację, losowe wycinanie (*cropping*) itp. W ramach przekształceń fotometrycznych można modyfikować jasność, kontrast czy nasycenie kolorów. Jedną z postawowych zasad, podczas stosowania *data augmentation* jest konieczność zachowania pierwotnych etykiet danych. Z tego powodu w przypadku zbioru *MNIST* nie można dokonywać transformacji *horizontal flipping*.



Rysunek 1: Oryginalny obrazek

Jako transformacje zdefiniowano:

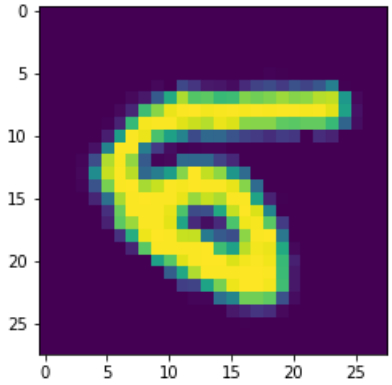
- przeskalowanie obrazka (i uzupełnienie zerami (*padding*) do rozmiaru obrazka wejściowego)



Rysunek 2: Obrazek przeskalowany

  Miara przeskalowania jest generowana z rozkładu jednostajnego z przedziału $[0.5,\ 0.9]$ z dokładnością do jednego miejsca po przecinku.

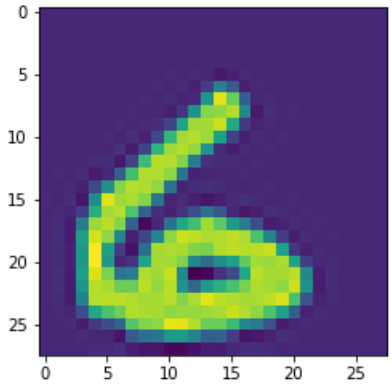
- rotację



Rysunek 3: Obrazek po zastosowaniu rotacji

Kąt rotacji generowany jest losowo z rozkładu jednostajnego $[-40,\ 40]$ stopni.

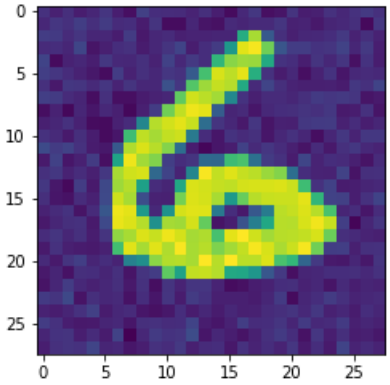
- translację połączoną ze zmniejszeniem ostrości obrazka



Rysunek 4: Obrazek po translacji wraz ze zmniejszeniem ostrości

Wartości w macierzy przesuwane są o wektor, którego współrzędne pochodzą z rozkładu jednostajnego $[-5,\ 5]$. Algorytm w trakcie wykonywania tej operacji dokonuje pewnej interpolacji rzędu od 0 do 5. Im wyższy jest rząd, tym obrazek jest bardziej rozmyty (co też stanowi użyteczną modyfikację w kontekście *data augmentation*), dlatego też rząd interpolacji również jest generowany z rozkładu jednostajnego.

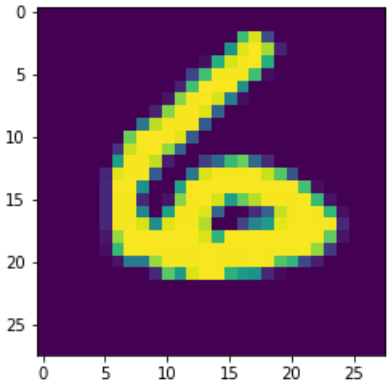
- dodanie szumu



Rysunek 5: Obrazek z dodanym szumem

Do obrazków dodano szum z rozkładu gaussa o parametrach (średnia = 0, odchylenie standardowe = 0.05)

- *ZCA whitening* (wybielanie)



Rysunek 6: Obrazek po zastosowaniu mechanizmu wybielania

Metoda polegająca na ekstrakcji poprzez rozkład według wartości osobliwych (SVD) macierzy charakteryzujących konkretny rekord, aby na ich podstawie wyliczyć macierz przekształcenia (w tej metodzie nie ma losowości).
ZCA whitening [1] jest liniową transformacją, która randomizuje wejściowy wektor:

$$x = (x_1,\ x_2,\ \ldots,\ x_n) \tag{14}$$

poprzez wykorzystanie macierzy kowariancji:

$$cov\,(x) = \ \Sigma \tag{15}$$

w ramach wyrażenia:

$$y = (y_1,\ y_2,\ \ldots,\ y_n)^T \ = \ W \cdot x \tag{16}$$

z zachowaniem własności, że:

$$cov\,(y) = I \tag{17}$$

Macierz W nazywana jest *whitening matrix* i musi spełniać warunek:

$$W^T \cdot W \;=\; \Sigma^{-1} \tag{18}$$

W przypadku tego przekształcenia zmiany nie są widoczne „gołym okiem".

3 Szczegóły implementacyjne

Użytkownik może zdefiniować sam konkretną topologię sieci (liczbę warstw ukrytych oraz liczbę neuronów na każdej warstwie) oraz następujące parametry:

- funkcje aktywacji dla każdej warstwy ukrytej (do wybou: *tanh*, *sigmoid*, $ReLU$)
- funkcja kosztu (do wyboru: *cross-entropy*, *mse*)
- regularyzacja
- *weight decay*
- *momentum*
- rozmiar *batcha*
- współczynnik uczenia (*ang. learning rate*)
- liczba epok

4 Wyniki

W celu porównania efektywności i poprawności sieci z różnymi parametrami napisano skrypt do automatycznych testów, który zbierał metryki w postaci *accuracy* do pliku w formacie *JSON*. Następnie w skrypcie `tests_analysis.ipynb` dokonano zestawienia wyników.

We wszystkich testach zgromadzonych w zestawieniu poniżej użyto *learning rate* o wartości 0.04, `batch_size` równy 128, *weight decay* równy 0.005 oraz momentum o wartości 0.9.

id	accuracy	epochs	error_function	hidden_activ_func	hidden_layer_num	img_augme	momentum	regularization
30	0.9744	30	cross_entropy	[relu] * 2	[128] * 2	True	False	False
21	0.9727	20	cross_entropy	[tanh] * 3	[128] * 3	True	False	False
1	0.9715	30	cross_entropy	[tanh] * 4	[128] * 4	False	True	False
16	0.9701	20	cross_entropy	[relu] * 2	[128] * 2	False	False	False
0	0.9695	30	cross_entropy	[tanh] * 2	[128] * 2	False	True	False
20	0.9692	20	cross_entropy	[tanh] * 3	[128] * 3	False	False	False
12	0.9686	20	cross_entropy	[tanh] * 2	[128] * 2	False	False	False
17	0.9684	20	cross_entropy	[relu] * 2	[128] * 2	True	False	False
26	0.9682	20	cross_entropy	[tanh] * 2	[128] * 2	False	True	False
4	0.9681	20	cross_entropy	[tanh] * 2	[128] * 2	False	True	False
24	0.9666	20	cross_entropy	[tanh] * 2	[128] * 2	True	False	False
31	0.9665	20	cross_entropy	[tanh] * 5	[128, 64, 32, 16, 8]	True	False	False
13	0.9659	20	cross_entropy	[tanh] * 2	[128] * 2	True	False	False
19	0.9600	20	cross_entropy	[tanh] * 1	[128] * 1	True	False	False
28	0.9592	20	cross_entropy	[tanh] * 3	[128] * 3	True	True	True
18	0.9579	20	cross_entropy	[tanh] * 1	[128] * 1	False	False	False
10	0.9571	10	cross_entropy	[tanh] * 2	[128] * 2	False	False	False
27	0.9536	20	cross_entropy	[tanh] * 2	[128] * 2	True	True	True
5	0.9519	20	cross_entropy	[tanh] * 10	[20] * 10	False	True	False
25	0.9514	20	cross_entropy	[tanh] * 2	[128] * 2	False	False	True
8	0.9510	20	cross_entropy	[tanh] * 5	[20] * 5	True	True	True
2	0.9504	30	cross_entropy	[tanh] * 2	[20] * 2	False	True	False
3	0.9498	20	cross_entropy	[tanh] * 5	[20] * 5	False	True	False
6	0.9368	20	cross_entropy	[tanh] * 20	[20] * 20	False	True	False
15	0.9220	20	cross_entropy	[sigmoid] * 2	[128] * 2	True	False	False
14	0.9205	20	cross_entropy	[sigmoid] * 2	[128] * 2	False	False	False
11	0.8842	5	cross_entropy	[sigmoid] * 2	[128] * 2	True	False	False
9	0.3596	5	mse	[tanh] * 2	[128] * 2	False	False	False
22	0.3387	20	mse	[tanh] * 2	[128] * 2	False	False	False
29	0.3167	20	mse	[tanh] * 2	[128] * 2	False	True	False
23	0.2888	20	mse	[tanh] * 2	[128] * 2	True	False	False
7	0.2110	20	cross_entropy	[tanh] * 30	[20] * 30	False	True	False

Najlepszą siecią w naszym przypadku (97.44%) okazała się sieć z dwoma warstwami ukrytymi z 128 neuronami w każdej i funkcjami aktywacji w postaci reLU. Sieć była trenowana przez 30 epok, a zbiór treningowy został powiększony o 1000 obrazków w ramach *data augmentation*. Niewiele gorszym wskaźnikiem *accuracy* skutkowało zastosowanie sieci z trzema warstwami ukrytymi (128 neuronów w każdej) z funkcjami aktywacji *tanh* trenowana przez *20* epok (97.27%). W tym wypadku także zastosowano *data augmentation*.

4.1 Wpływ parametrów sieci na wartosć *accuracy*

Poniżej zostało przedstawione, jaki wpływ na wartość *accuracy* ma dobór odpowiednich parametrów sieci.

4.1.1 Wpływ różnych funkcji aktywacji

Testy o identyfikatorach 16 i 12 różniły się jedynie rodzajem zastosowanych funkcji aktywacji W sieci o numerze 16 zastosowano we wszystkich warstwach *reLU*, natomiast w 12 – tangens hiperboliczny. Okazało się, że w przypadku zastosowania funkcji *reLU* wartość *accuracy* była o 0.15 pkt. procentowego wyższa niż w przypadku funkcji *tanh*. Można więc stwierdzić, że różnica jest znikoma. Analogiczny test (14) został przeprowadzony dla sieci o takich samych parametrach jednak z funkcją aktywacyjną *sigmoid*. W tym przypadku *accuracy* było aż o 4.96 pkt. procentowego niż dla analogicznej sieci z funkcją *reLU*.

W zaimplementowanej przez nas sieci neuronowej funkcja *sigmoid* skutkuje znacznie mniejszą wartością *accuracy* niż *reLU* i *tanh*.

4.1.2 Wpływ zastosowania w algorytmie SGD momentum

Porównując testy z tabeli o oznaczeniach 12 i 26, można stwierdzić, że różnią się jedynie tym, że w przypadku testu 26 zastosowano *momentum*. Sieć była trenowana przez 20 epok, używając jako funkcji kosztu *cross entropy*. Składała się z dwóch warstw ukrytych o 128 neuronach. Użyto funkcji aktywacji *tanh*. Okazało się, że w przypadku użycia *momentum* (test 26) sieć osiągnęła minimalnie wyższe (o 0.04 pkt. procentowego) *accuracy*. Niemniej jest to wielkość tak mała, że można uznać, iż w tym konkretnym przypadku wpływ zastosowania w sieci *momentum* na poprawność klasyfikacji jest znikomy.

4.1.3 Wpływ zastosowania data augmentation

Testy o numerach 20 i 21 różnią się jedynie tym, w pierwszym przypadku zastosowano *data augmentation*. Ta też sieć skutkowała *accuracy* lepszym o 0.35 pkt. proc. Warto podkreślić, że w ramach *data augmentation* zmodyfikowano jedynie 1 000 obrazków (ze względu na zasoby obliczeniowe). W przypadku większego odsetka przekształconych obrazków, można by było oczekiwać większej poprawy *accuracy*.

4.1.4 Wpływ regularyzacji

Testy o numerach 25 i 12 różnią się jedynie tym, że w przypadku tego pierwszego zastosowano *regularyzację*. Niemniej okazało się, że zastosowanie czynnika *weigth decay* pogorszyło zdolność sieci do poprawnej klasyfikacji o 1.72 pkt. procentowego.

4.1.5 Wpływ liczby epok

Porównano wynik błędu klasyfikacji dla sieci z dwoma warstwami ukrytymi (128 neuronów w każdej warstwie) z funkcjami aktywacji – *tanh*, bez zastosowania *regularyzacji*, *momentum*, ani *data augmentation* dla różnej liczby epok treningowych.

Rysunek 7: Wykres porównujący wartość *accuracy* ze względu na liczbę epok

Wyraźnie widać, że dla analizowanej architektury trend jest spadkowy, czyli im więcej epok tym mniejszy błąd klasyfikacji.

4.1.6 Wpływ liczby warstw

Analizowane architektury sieci miały po 20 neuronów w warstwach ukrytych. Jako funkcji aktywacji użyto *tanh*. Sieci byłby trenowane prze 20 epok – z wykorzystaniem *momentum*.

Rysunek 8: Wykres porównujący wartość *accuracy* ze względu na liczbę warstw

Można zaobserwować, że użycie 5, 10, 20 warstw ukrytych skutkowało ponad 90% *accuracy*, podczas gdy zastosowanie 30 warstw wiązało się z dużym spadkiem poprawności klasyfikacji do wartości jedynie ok. 20%.

4.1.7 Wpływ liczby neuronów

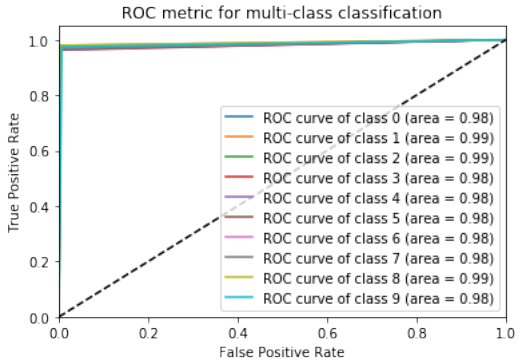
Porównano sieć o identyfikatorach równych 2 i 26. Obie byłby trenowane przez 20 epok z zastosowaniem *momentum*. Pierwsza z nich miała dwie warstwy ukryte składające się z 20 neuronów, natomiast druga – 128 neuronów. Mniejsza sieć cechowała się gorszą o 1.78 pkt. wartością *accuracy*.

W następnych podrozdziałach zaprezentowano szczegółowe wartości metryk dla sieci, która skutkowała najlepszą wartością accuracy.

4.2 Wykres *ROC*

Poniżej został przedstawiony wykres *ROC*, na który zostały nałożone dane dotyczące każdej klasy. Oddzielne wykresy dla każdej z klas zostały zawarte w pliku `tests_analysis.ipynb`. Na

wykresie poniżej widać, że dla każdej z kategorii pole powierzchni pod krzywą ROC wynosi niemal wartość maksymalną równą 1 (która odpowiada idealnym klasyfikatorom). Warto też podkreślić, że kształt krzywych ROC jest praktycznie linią prostą (nie jest poszarpana).



Rysunek 9: Wykres ROC dla wszystkich klas

4.3 Macierz pomyłek

Poniżej została przedstawiona macierz pomyłek dla „najlepszej" sieci

	0	1	2	3	4	5	6	7	8	9
0	964	0	0	0	0	6	4	2	2	2
1	0	1124	2	1	0	1	2	1	4	0
2	5	1	1004	6	1	0	3	7	5	0
3	1	0	5	984	0	4	0	5	6	5
4	3	0	5	0	955	0	2	3	1	13
5	2	0	0	10	1	865	7	1	5	1
6	5	3	2	1	3	6	936	0	2	0
7	2	6	8	2	0	0	0	997	2	11
8	2	0	6	6	3	4	3	3	945	2
9	5	1	2	9	10	1	0	7	4	970

Rysunek 10: Macierz pomyłek dla „najlepszej" sieci

Można wywnioskować, że najwięcej pomyłek było przy próbie klasyfikacji obrazka, na którym była zapisana czwórka i został on zaklasyfikowany jako dziewiątka (13 błędów tego rodzaju). Również, w odwrotny sposób – klasyfikator 10 razy sklasyfikował dziewiątkę jako czwórkę.

4.4 Raport klasyfikacji

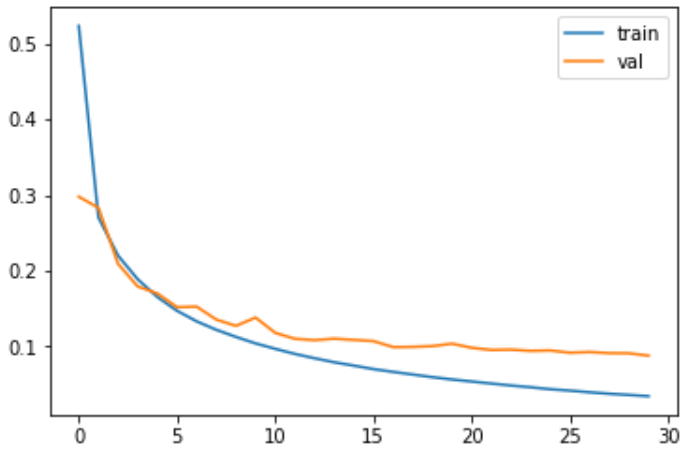
Poniżej znajdują się obliczone metryki *precision*, *recall* oraz *f1* dla każdej klasy osobno. Wyniki zostały sporządzone w oparciu o sieć dającą najwyższe *accuracy*.

klasa	precision	recall	f1
0	0.97	0.98	0.98
1	0.99	0.99	0.99
2	0.97	0.97	0.97
3	0.97	0.97	0.97
4	0.98	0.97	0.98
5	0.98	0.97	0.97
6	0.98	0.98	0.98
7	0.97	0.97	0.97
8	0.97	0.97	0.97
9	0.97	0.96	0.96

Tabela 1: Raport klasyfikacji dla „najlepszej" sieci

4.5 Wykresy funkcji kosztu

Poniżej przedstawiono wykres funkcji kosztu dla sieci uzyskującej najwyższe *accuracy*. Wyraźnie widać, że wartości obu funkcji maleją, a więc nie występuje zjawisko przeuczenia.

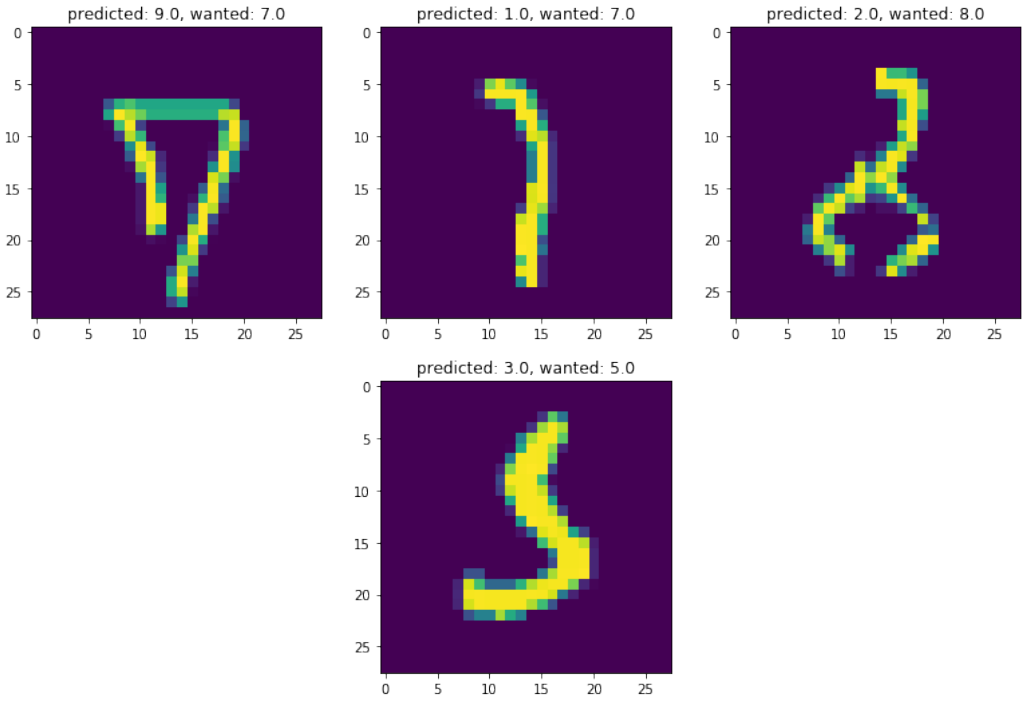


Rysunek 11: Wykres funkcji kosztu dla sieci o najwyższym *accuracy*

Wykresy funkcji kosztu dla innych sieci zostały zamieszczone w *notebooku* z zawartym kodem i procedurą testowania sieci.

4.6 Źle sklasyfikowane obrazki

Sieć o największym *accuracy* sklasyfikowała niepoprawnie łącznie 256 obrazków. Wykaz wszystkich obrazków został zawarty w pliku `tests_analysis.ipynb`. Poniżej znajduje się kilka wybranych przykładów.



Rysunek 12: Wybrane źle sklasyfikowane obrazki przez „najlepszą” sieć

5 Wnioski

Na podstawie analizy wyników uzyskanych przez różne topologie sieci neuronowych zauważono, że 30-sto czy 5-cio warstwowe skutkowały znacznie mniejszym *accuracy* niż sieci płytkie. Podejrzewa się, że wynika to z wystąpienia zjawiska *vanishing gradient*, który polega na tym, że w trakcie propagacji wstecznej przez wiele warstw wielokrotnie mnoży się gradienty o małych (często mniejszych od zera wartości), co powoduje, że dążą one do zera. Natomiast, w sytuacji, gdy gradienty są zerowe sieć przestaje mieć zdolność do aktualizacji wag w danej warstwie, a więc proces uczenia znacznie „wyhamowywuje”.

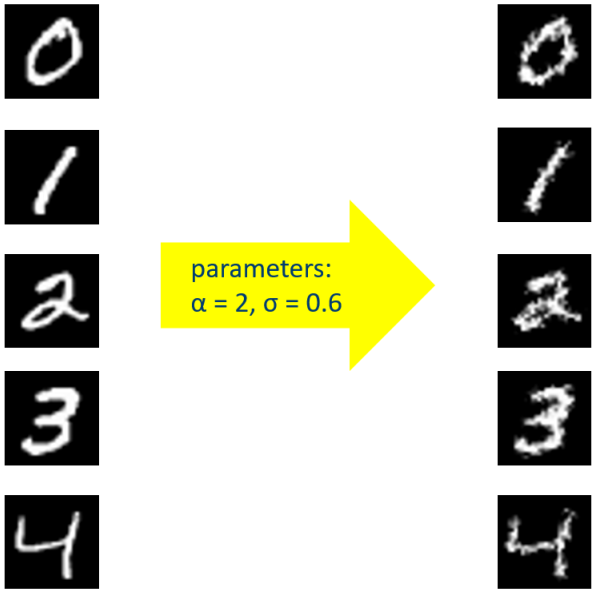
Zdecydowanie niższą wartością *accuracy* skutkowało użycie jako funkcji kosztu – *MSE*. Wynik ten pokrywa się z przewidywaniami, bowiem *MSE* jest zazwyczaj stosowane w zadaniu regresji, natomiast *cross entropy* w zadaniach klasyfikacji.

6 Możliwe kierunki rozwoju projektu

Na *perfomance* sieci neuronowej wpływa wiele czynników. Jedną z kluczowych decyzji jest wybór optymalizatora. W naszym rozwiązaniu zastosowano algorytm *SGD*, niemniej warto by było przetestować jeszcze algorytm *Adam* czy *RMSProp*, które z uwagi na bardziej wyszukane formuły pozwalają na efektywniejszy trening sieci. W przypadku zastosowania algorytmu *Adam* mniej istotny staje się dobór hiperparametru *learning rate*, ponieważ jego wartość jest dostosowywana na bieżąco na podstawie przebiegu procesu uczenia. Warto także także wzbogacić algorytm o kryterium stopu, które w sytuacji gwałtownego wzrostu błędu (np. trzykrotnego) na zbiorze walidacyjnym zakończy trening i wróci do poprzedniego stanu wag, takie rozwiązanie wymagałoby przechowywania w ramach działania programu w każdej iteracji zestawu wag z poprzedniej epoki.

Ciekawym zagadnieniem byłoby także zastosowanie *data augmentation* w postaci *elastic transformation* [3]. W ramach tego przekształcenia symuluje się efekty wynikająca z różnych skurczy mięśni dłoni. Transformacja ma charakter dwuparametrowy (α, σ). Pierwszy z nich

określa miarę zniekształcenia, natomiast drugi „gładkość" transforamacji. Poniżej zaprezentowano porównanie obrazków przed i po zastosowaniu transformacji.



Rysunek 13: *Elastic transformation*, Opracowanie własne

7 Bibliografia

[1] Agnan Kessy, Alex Lewin, Korbinian Strimmer, 'Optimal whitening and decorrelation', 2016
[2] Christopher M. Bishop, 'Pattern Recognition and Machine Learning (Information Science and Statistics)', 2006
[3] Partice Y. Simard, Dave Steinkraus, John C. Platt, 'Best practices for convolutional neural networks applied to visual document analysis', 2003
[4] Xavier Glorot and Yoshua Bengio, chapter: Understanding the difficulty of training neural networks, 'Proceedings of the Thirteenth International Conference n Artificial Intelligence and Statistics', pages 249-256; 2010
[5] http://neuralnetworksanddeeplearning.com