Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **INF-PPT**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Uczenie maszynowe - analiza wybranego zagrożenia

Jakub Kazimierski

Opiekun pracy
dr hab. inż. Łukasz Krzywiecki

Słowa kluczowe: szyfrowanie homomorficzne, zagrożenie prywatności, sieci neuronowe

Wrocław 2022

Streszczenie

Celem pracy jest: zastosowanie szyfrowania homomorficznego na modelach sieci neuronowych, w celu zapewnienia bezpieczeństwa prywatności w zewnętrznych środowiskach wykonawczych jakimi są chmury obliczeniowe.

Na danych ze zbioru MNIST, wytrenowano prosty w budowie model liniowy sieci neuronowej i bardziej złożony model konwolucyjnej sieci neuronowej o nazwie LeNet1. Dane z treningów i testów zebrano w formie wykresów. Modele wytrenowane zostały poddane szyfrowaniu homomorficznemu, a następnie przetestowano je na zaszyfrowanych danych ze zbioru MNIST. Dane z testów zebrano w formie wykresów. Porównano wyniki precyzji oraz czasu działania uzyskane przez sieci zaszyfrowane i sieci niezaszyfrowane. Na bazie wyników wyciągnięto odpowiednie wnioski.

Abstract

Goal of this engineer thesis, is to use homomorphic encryption at neural network models in order to keep privacy at external environments like computing clouds.

At data from MNIST database was trained: simple linear neural network model and more complex convolutional neural network model LeNet1. Data from trainings and tests was gathered into charts. Homomorphic encryption was used at trained models and encrypted models were tested on encrypted data from MNIST database. Data from tests were gathered into charts. Results from model encrypted and models not encrypted were compared and based on this comparision conclusions were made.

Spis treści

Spis rysunków

Spis tabel

Wstęp

Uczenie maszynowe jest obszarem sztucznej inteligencji, opiera się na obliczeniach których zadaniem jest minimalizowanie błędu wyniku, poprzez analizę błędów otrzymywanych podczas nauki. Proces nauki odbywa się na zbiorach uczących, które zawierają próbne dane, dzięki którym algorytmy będą w stanie rozpoznać dane docelowe. Trening służy do wyuczenia modelu uczenia maszynowego określonego celu np. prognozowania, podejmowania decyzji bądź rozpoznawania obiektów na obrazach. Modele na których wykonywane są obliczenia to sieci neuronowe, których zadaniem jest imitować neurony ludzkiego mózgu. Na ich budowę składają się neurony przechowujące wartości liczbowe oraz połączenia między nimi, do których przypisana jest waga liczbowa. Obliczenia dokonywane na wartościach w neuronach i wagach, zwracają wyniki pozwalające na sklasyfikowanie danych wejsciowych poprzez wartość lub wartości wyjsciowe. Miejscem pracy takiej sieci może być maszyna lokalna lub udostępniony w tym celu serwer chmurowy, którego zasoby pamięciowe, pozwalają na odciążenie maszyn lokalnych oraz na udostępnienie użytkowania z sieci neuronowej, szerszemu gronu użytkowników. Przesłany do chmury model, jest w większym stopniu podatny na zarożenie kradzieży, niż model przetrzymywany lokalnie. Atakujący może próbować wykraść wytrenowaną sieć, lub odkryć strukturę modelu w stopniu pozwalającym na jego przybliżone odwzorowanie. Istnieją jednak sposoby pozwalające na eliminację tego zagrożenia. Jednym z nich jest proces szyfrowania homomorficznego. Pozwala on na zaszyfrowanie danych wejściowych, wyjsciowych, oraz wag i neuronów sieci, w taki sposób, aby możliwym było otrzymanie poprawnego wyniku na danych zaszyfrowanych. Poznanie prawdziwej wartości zaszyfrowanego elementu nie będzie wykonalne bez posiadania odpowiedniego klucza deszyfrującego. Sprawia to, że w przypadku kradzieży zaszyfrowanego homomorficznie modelu, niewykonalnym będzie jego wykorzystanie, a także niewykonalnym staje się poznanie struktury sieci. **Zadaniem** tej pracy jest zastosowanie szyfrowania homomorficznego na wytrenowanych sieciach neuronowych w celu zapewnienia bezpieczeństwa prywatności w zewnętrznych środowiskach wykonawczych.

Zakres pracy obejmuje: zaimplementowanie dwóch modeli sieci neuronowych różniących się budową, wytrenowanie ich w celu rozpoznawanie obrazów pisma odręcznego. Porównanie precyzji i czasu działania tych sieci podczas treningów i testów. Zaszyfrowanie tych modeli sieci oraz obrazów, które sieci będą rozpoznawać, w sposób homomorficzny. Przeprowadzenie testów na modelach zaszyfrowanych. Porównanie precyzji i czasu działania, między modelami zaszyfrowanymi i niezaszyfrowanymi. Wyciągnięcie wniosków z uzyskanych wyników.

Celem projektu jest:

- wytrenowanie sieci neuronowych na rozpoznawania obrazów pisma odręcznego,
- porównanie precyzji i czasu działania sieci różniących się budową,
- zaimplementowanie szyfrowania homomorficznego w celu ochrony sieci neuronowej przed atakami kradzieży,
- sprawdzenie różnic precyzji i czasu działania między sieciami szyfrowanymi i nieszyfrowanymi,
- analiza uzyskanych wyników oraz przedstawienie odpowiednich wniosków.

Zawartość dokumentu składa się z trzech rozdziałów. W rozdziale pierwszym omówiono zbiór danych na których uczyć się i operować będą sieci neuronowe. Przedstawiono wybrane do testów modele sieci

wraz z ich budową. Przeanalizowano proces przetwarzania obrazów przez użyte sieci oraz sposób interpretacji danych wyjściowych. Omówiony został użyty schemat szyfrowania homomorficznego. W rozdziale drugim przedstawiono skrypty użyte do wykonania eksperymentów. Opisane zostały technologie implementacji projektu: język programowania i biblioteki. Przedstawiono proces szyfrowania sieci neuronowych. Przeanalizowano proces działania sieci zaszyfrowanych. Omówiono przebieg testów przeprowadzanch na modelach sieci. Przedstawiono fragmenty kluczowych kodów źródłowych. W rozdziale trzecim ukazane zostały wyniki przeprowadzonych testów oraz ich graficzne reprezentacje. Na podstawie uzyskanych wyników przedstawione zostały wnioski.

Rozdział 1

Analiza problemu

W niniejszym rozdziale przytoczona została literatura związana z tematem pracy. Omówiono zagrożenie jakim jest możliwość kradzieży modelu sieci z chmury obliczeniowej. Przedstawiono dane użyte do zbudowania sieci neuronowych oraz pokazano ich architekturę. Szczegółowo zobrazowano użyty schemat szyfrowania homomorficznego, a następnie przedstawiony został proces szyfrowania danych za jego pomocą.

1.1 Przegląd literatury związanej z tematem pracy

Zagrożenie omawiane w pracy wiąże się z niebezpieczeństwem kradzieży wytrenowanego modelu sieci neuronowej. Sposoby kradzieży podzielono na dwa przypadki. Jednym z nich jest bezpośredni atak na model i przejęcie go. Sytuacja taka może mieć miejsce, gdy model umieszczony został w publicznym środowisku wykonawczym, bez weryfikacji dostępu użytkowników. Drugim przypadkiem jest atak mający na celu poznanie budowy sieci, na przykład poprzez poznanie danych trenignowych. Z tego powodu coraz częściej dane zostają utajnione w celu zapobiegania takim atakom. Jednak wraz z rozwojem działań prewencyjnych, rozwijają się też sposoby ataków, jednym z nich jest atak opisany w artykule [12]. Za jego pomocą atakujący jest w stanie zbudować model o zbliżonej precyzji do modelu atakowanego, poprzez analizowanie wartości zwracanych przez sieć neuronową. Cechą wspólną podatności na te ataki, jest jawność danych na których sieć operuje i które zwraca, a także wag połączeń między neuronami.

Czynnikiem eliminującym tę podatność jest wspomniane wcześniej szyfrowanie homomorficzne. Standardy takiego szyfrowania opisane zostały w artykule [11], pojęcie szyfrowania homomorficznego zobrazowano w artykue [9]. Dostępnych jest kilka standardów, użyty w tej pracy został schemat BrakerskiFan-Vercauteren ze względu na jego dostępność w bibliotece użytego języka programowania.

Budowa modelu uczenia maszynowego uwarunkowana jest zbiorem danych, na których sieć ma za zadanie przeprowadzać obliczenia. Na potrzeby pracy użyty został zbiór obrazów pisma odręcznego MNIST [7]. Są to obrazy o wymiarach 28 pikseli na 28 pikseli, które przedstawiają cyfry od zera do dziewięciu.

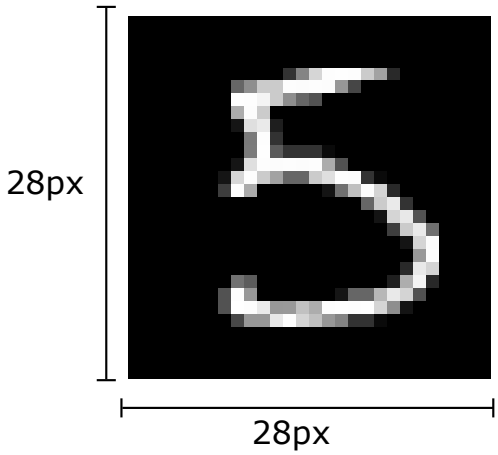
W projekcie zaimplementowane zostały dwa modele sieci. Pierwszy z nich to prosty model sieci gęstej posiadający trzy warstwy. Drugi model bazuje na schemacie LeNet, który został opisany w artykule [13]. Celem pracy jest zastosowanie schematu szyfrowania homomorficznego, dlatego dla uproszczenia przeprowadzanych obliczeń użyty został schemat LeNet-1, który cechuje łatwa w implementacji budowa.

Metody działania sieci konwolucyjnych korzystających z metod próbkowania do których należy model LeNet-1 opisano w artykule [10].

1.2 Dane do uczenia maszynowego

Dane których użyto do wytrenowania i testowania sieci pochodzą ze zbioru MNIST. Są to obrazy cyfr pisma odręcznego. Zbiór trenujący posiada 60 000 modeli, a zbiór testujący 10 000. Obrazy posiadają wymiary 28 pikseli na 28 pikseli, które zawierają wartości od 0 symbolizującego kolor biały do 255 symbolizujących kolor czarny. Piksele z takich obrazków, w postaci macierzy o wymiarach 28 na 28 będą

stanowić wejściową warstwę do sieci neuronowej. Warstwę wyjściową sieci będą stanowić neurony zawierające wartości liczbowe, z których największa wskazuje numer neuronu w warstwie odpowiadający jednej z 10 klas bazy MNIST, są to odpowiednio: 0 - klasa oznaczająca cyfrę 0, 1 - klasa oznaczająca cyfrę 1, 2 - klasa oznaczająca cyfrę 2 itd. Przykład cyfry ze zbioru MNIST załączono na grafice 1.1



Rysunek 1.1: Odręcznie pisana cyfra ze zbioru MNIST posiadająca etykietę 5.

1.3 Architektura sieci

W pracy zaimplementowane zostały dwa modele sieci wytrenowane do rozpoznawania obrazów pisma odręcznego. Trening odbył się na zbiorze treningowym MNIST zawierającym 60 000 obiektów. Testy odbyły się na zbiorze testowym zawierającym 10 000 obiektów. Celem testów było porównanie precyzji oraz czasów działania sieci neuronowych.

1.3.1 Model liniowy

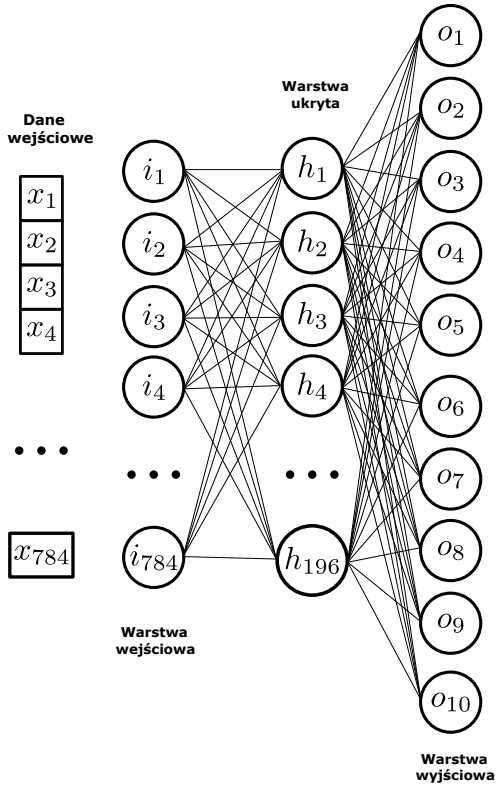
Model pierwszy to sieć posiadająca:

- warstwę spłaszczającą dane wejsciowe (nie jest to warstwa, która wykonuje obliczenia na danych),
- warstwę posiadająca 784 wejścia i 196 wyjść (Warstwa wejściowa),
- warstwę posiadająca 196 wejść i 10 wyjść (Warstwa ukryta),
- warstwę wyjściową składającą się z 10 neuronów.

Sieć jest gęsta, co znaczy, że wszystkie neurony z sąsiednich warstw są ze sobą połączone. Pozwala to na określenie struktury tablicy wag, łączących kolejne warstwy sieci, są to kolejno:

- 784 wiersze, po 196 elementów stanowiących wartości wag, używanych przez sieć neuronową,
- 1 wiersz, po 196 elementów stanowiących dodatkowe wartości,
- 196 wierszy, po 10 elementów,
- 1 wiersz, po 10 elementów.

Schemat tej sieci został przedstawiony na grafice 1.2



Rysunek 1.2: Schemat liniowej sieci neuronowej.

1.3.2 Model LeNet1

Model drugi został stworzony na bazie schematu LeNet-1. To sieć konwolucyjna, której autorem jest Yann Le-Cun, zbudowana pod koniec lat dziewięćdziesiątych ubiegłego wieku. Zadaniem sieci było rozpoznawanie obrazów o wymiarach 28px na 28px. Na budowę modelu składają sie:

- warstwa konwolucyjna posiadająca 4 mapy funkcji konwolucyjnej, o wymiarach 24 na 24,
- warstwa próbkująca wykonująca funkcję próbkowania na 4 mapach z poprzedniej powłoki i w rezultacie posiada 4 mapy o wymiarach 12 na 12,
- warstwa konwolucyjna posiadająca 3 mapy funkcji konwolucyjnej, którą wykonuje na otrzymanych 4 warstwach z poprzedniej powłoki, co skutkuje ilością 12 map o wymiarach 8 na 8,
- warstwa próbkująca wykonująca funkcję próbkowania na 12 mapach z poprzedniej powłoki i w rezultacie posiada 12 map o wymiarach 4 na 4,
- warstwa spłaszczająca dane wejsciowe w formę tablicy jednowymiarowej,
- warstwa liniowa posiadająca 196 wejść i 10 wyjść,
- warstwa wyjściowa składającą się z 10 neuronów.

Sieć konwolucyjną, charakteryzują dwie funkcje wykonywane przez warstwy tej sieci. Są to:

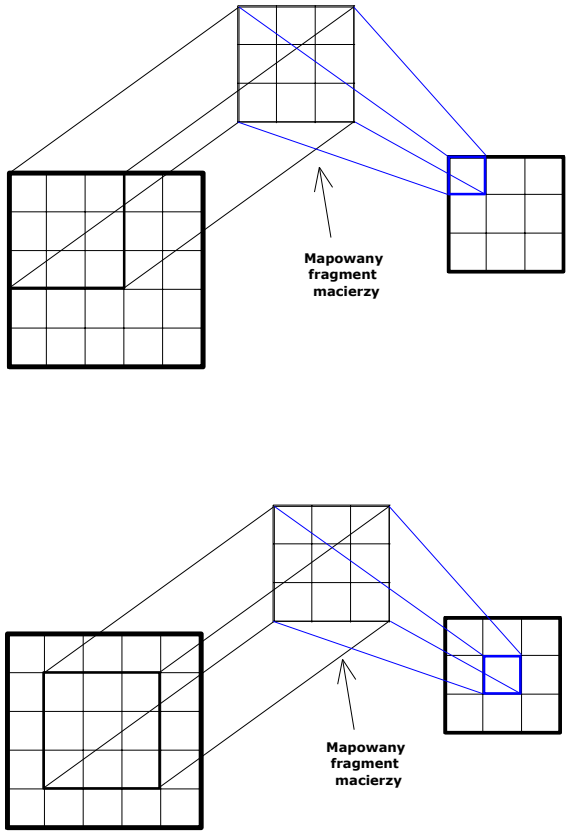
- Operacja konwolucji, zachodzącej na dwóch macierzach. Matematyczną formułę tej operacji możemy zapisać w postaci:

$$\begin{bmatrix} x_{11} & x_{12} & ... & x_{1n} \\ x_{21} & x_{22} & ... & x_{2n} \\ ... & ... & ... & ... \\ x_{m1} & x_{m2} & ... & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & ... & y_{1n} \\ y_{21} & y_{22} & ... & y_{2n} \\ ... & ... & ... & ... \\ y_{m1} & y_{m2} & ... & y_{mn} \end{bmatrix} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{(m-i,n-j)} y_{(1+i,1+j)}$$

gdzie:

- ***** oznacza mnożenie,
- macierz pierwsza to fragment mapowanej macierzy weejściowej o wymiarach m na n,
- macierz druga oznacza przetwarzany przez filtr mapujący fragment macierzy wejściowej.

Wizualizację tej operacji przedstawiono na grafice 1.3



Rysunek 1.3: Wizualizacja operacji konwolucji.

- Operacja próbkowania

  Operacja ta polega na przetwarzaniu wartości filtrowanego fragmentu macierzy w pojedynczą wartość, która zostaje zamieszczona jako komórka macierzy powstałej na skutek tej operacji. Matematycznie można przedstawić próbkowanie jako:

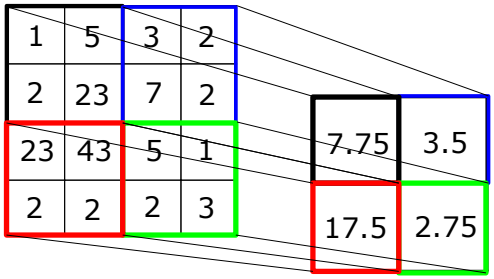
$$O_{i,j} = f(I_{k,l}, \quad S*i \leqslant k < S*i+m, \quad S*j \leqslant l < S*j+m)$$

  gdzie:

  - O oznacza macierz wyjściową,

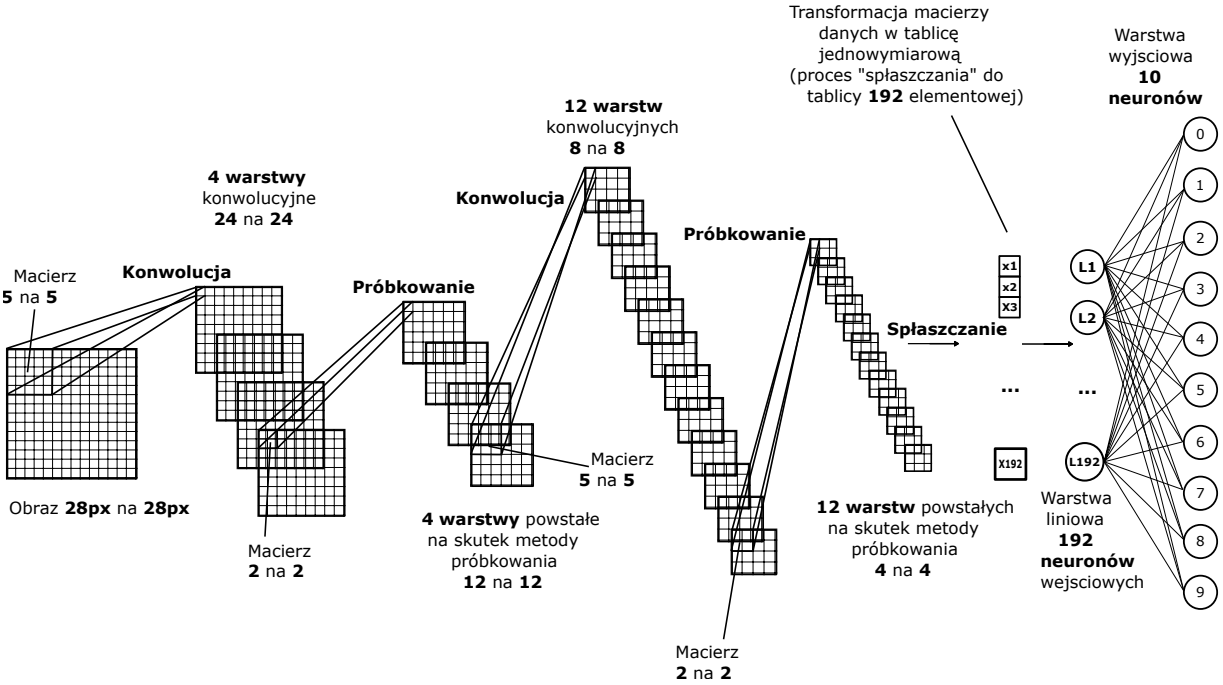
– I oznacza macierz wejściową,
– S oznacza numer kroku operacji,
– m oznacza wymiar filtra o rozmiarze m x m,
– i, j oznaczają współrzędne w wierszu i kolumnie macierzy wyjściowej,
– k, l oznaczają współrzędne macierzy wejściowej,
– $f()$ oznacza funkcję matematyczną,
– $*$ oznacza operację mnożenia.

Należy zaznaczyć, że występują różne rodzaje funkcji f wykonywanej podczas operacji próbkowania. Są to: wybór maksymalnej wartości, obliczenie średniej. W sieci LeNet1 wykorzystana jest operacja próbkowania wyliczająca średnią wartość liczbową z filtrowanego fragmentu macierzy. Wizualizację tej operacji przedstawiono na grafice 1.4



Rysunek 1.4: Wizualizacja operacji próbkowania.

Aby zobrazować schemat sieci LeNet1 przytoczona została grafika 1.5



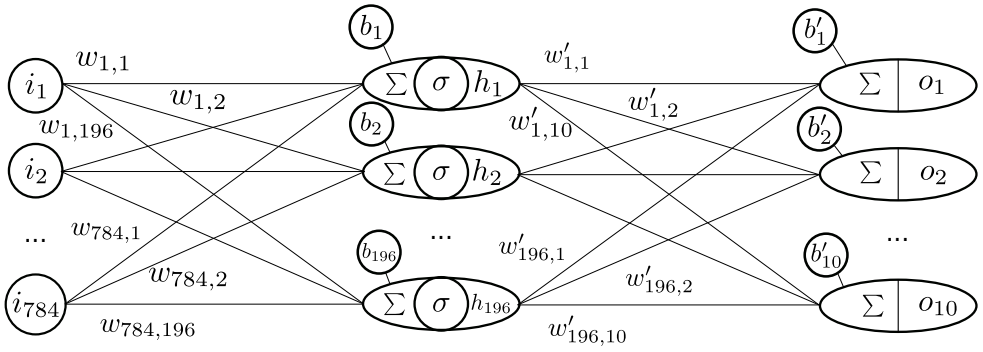
Rysunek 1.5: Schemat sieci LeNet1.

1.4 Działanie sieci

W tym podrozdziale omówiony został schemat działania przytoczonych uprzednio modeli sieci neuronowych, na danych ze zbioru obrazów cyfr pisma odręcznego MNIST. Obrazy są zapisane w postaci macierzy o wymiarach 28 na 28 i każdy element posiada wartości z zakresu od 0, które oznacza kolor biały, do 255 które oznacza kolor czarny. Dla usprawnienia obliczeń wykonywanych przez warstwy sieci, wartości te zostają przeskalowane do wartości z zakresu od 0 do 1.

1.4.1 Model liniowy

Pierwsza omawiana sieć, to model stworzony z połączonych ze sobą warstw liniowych. Powłoka pierwsza posiada 784 wejścia, co wynika z wymiarów tablicy danych jaką przyjmuje. Macierz o wymiarach 28 na 28 zostaje przetransformowana przez warstwę spłaszczającą w macierz jednowymiarową posiadającą 784 elementy. Wartości, które są w niej przechowywane, należą do zakresu od 0 do 1. Schemat działania pierwszego modelu sieci neuronowej został przedstawiony na grafice 1.6



Rysunek 1.6: Schemat działania liniowej sieci neuronowej.

Dane które zostają przekazane do warstwy wejściowej, są następnie przetwarzane za pomocą wag przypisanych do połączeń między warstwami. Dla zobrazowania tego procesu, przedstawiono matematyczną formułę uzyskania wartości, która zostaje przekazana do pierwszego neuronu warstwy ukrytej. Zgodnie z oznaczeniami na grafice:

$$h_1 = \sigma(\sum_{j=1}^{784}(i_j * w_{j,1}) + b_1)$$

gdzie:

- h_1 oznacza wartość przekazaną do pierwszego neuronu powłoki ukrytej,
- σ oznacza funkcję aktywującą, w tym modelu jest to tanges hiperboliczny,
- i oznacza wartość neurona o numerze j z powłoki wejściowej,
- $w_{j,1}$ oznacza wagi neuronów z powłoki wejściowej przypisane do neurona h_1,
- b_1 oznacza bias, który jest wartością stałą,
- * oznacza operację mnożenia.

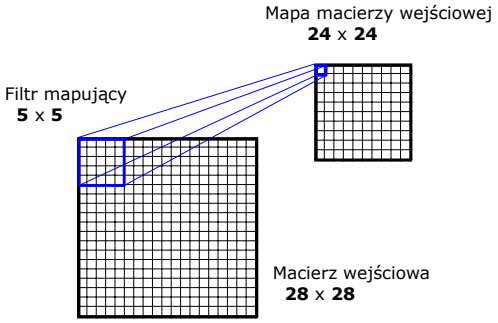
W sposób analogiczny wartości są przekazywane do pozostałych neuronów tej warstwy. Schemat obliczania wyników dla ostatniej warstwy (wyjściowej), różni się brakiem funkcji aktywującej. Matematyczny zapis funkcji obliczającej wartość dla pierwszego neuronu z tej warstwy ma wtedy postać:

$$O_1 = \sum_{j=1}^{192}(h_j * w'_{j,1}) + b'_1$$

Analogicznie wyliczane są pozostałe wartości wyjściowe. Wartość największa symbolizuje etykietę cyfry, którą miał rozpoznać model.

1.4.2 Model LeNet1

Model drugi to sieć LeNet1. Na grafice 1.7 zobrazowano operację konwolucji, z pierwszego fragmentu macierzy danych wejściowych, do pierwszej komórki macierzy reprezentującej pierwszą mapę tej funkcji.



Rysunek 1.7: Schemat operacji konwolucji.

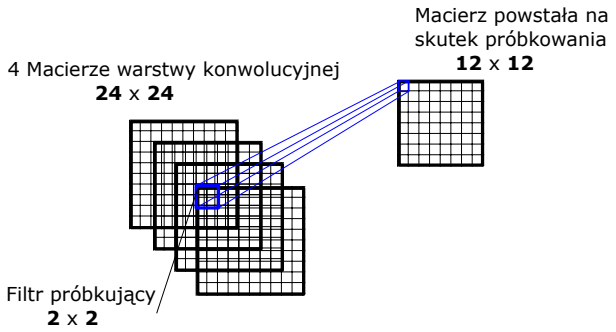
W tej operacji wykorzystywane są 4 filtry mapujące macierz wejściową, co skutkuje ilością 4 macierzy w pierwszej warstwie konwolucyjnej. Matematycznie formułę tej operacji można zapisać jako:

$$o_{i,j}^{(s)} = \sigma(b_{i,j}^{(s)} + \sum_{l=0}^{4}\sum_{m=0}^{4} k_{l,m}^{(s)} * a_{i+l,j+m})$$

gdzie:

- filtr mapujący posiada wymiary 5 na 5,
- $o_{i,j}^{(s)}$ oznacza wartość przekazaną do mapy funkcji, o numerach współrzędnych i, j, oraz numerze mapy s,
- σ oznacza funkcję aktywującą, w tym modelu jest to tanges hiperboliczny,
- $k_{l,m}^{(s)}$ oznacza wartość komórki filtra mapującego, o numerach współrzędnych l, m, oraz numerze mapy s,
- $a_{i+l,j+m}$ oznacza wartość z macierzy wejściowej, o numerach współrzędnych $i + l, j + m$,
- $b_{i,j}^{(s)}$ oznacza bias, który jest wartością stałą,
- * oznacza operację mnożenia.

Analogicznie operacja konwolucji przebiega dla kolejnych warstw mapujących. Procesem który zachodzi po konwolucji, jest próbkowanie wyliczające średnią wartość liczbową z filtrowanego fragmentu macierzy wejściowej. Na grafice 1.8 zobrazowano operację próbkowania z pierwszej warstwy powstałej na skutek konwolucji.



Rysunek 1.8: Schemat operacji próbkowania.

Matematyczny zapis:

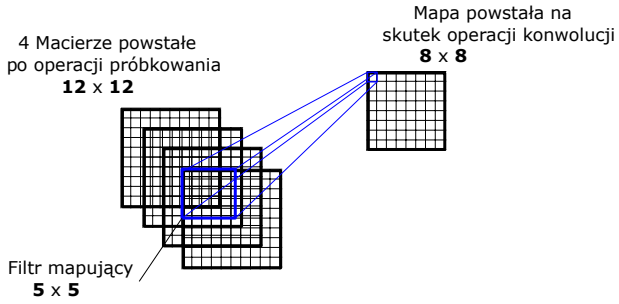
$$o_{i,j}^{w} = \frac{\sum_{k=S*i}^{S*i+m} \sum_{l=S*j}^{S*j+m} (i_{k,l}^{w})}{m * m}$$

gdzie:

- $o_{i,j}^{w}$ oznacza wartość w macierzy wyjściowej o numerze w, o współrzędnych i, j,
- $i_{k,l}^{w}$ oznacza wartość w macierzy wejściowej o numerze w, o współrzędnych k, l,
- S oznacza numer kroku operacji,
- m oznacza wymiar filtra o rozmiarze m x m,
- $*$ oznacza operację mnożenia.

W tej operacji filtrowana jest każda macierz z pierwszej warstwy konwolucyjnej, co skutkuje uzyskaniem również 4 macierzy w warstwie próbkującej.

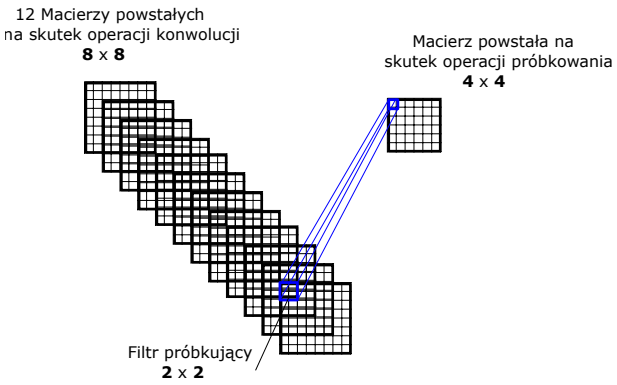
Na warstwie powstałej na skutek próbkowania, zachodzi ponownie operacja konwolucji, schemat ukazany na grafice 1.9



Rysunek 1.9: Schemat drugiej operacji konwolucji.

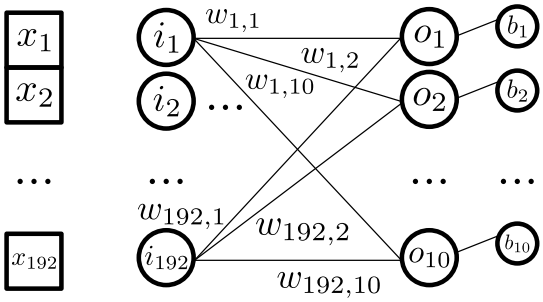
W tej operacji wykorzystane są 3 filtry mapujące dla każdej z filtrowanych macierzy, co skutkuje ilością 12 macierzy w warstwie konwolucyjnej.

Na warstwie powstałej na skutek drugiej operacji konwolucji, zachodzi ponownie operacja próbkowania, na skutek czego w kolejnej warstwie próbkującej znajduje się 12 macierzy, schemat ukazany na grafice 1.10



Rysunek 1.10: Schemat drugiej operacji próbkowania.

Następnie format danych z warstwy powstałej na skutek drugiej operacji próbkowania, zostaje przekształcony w tablicę jednowymiarową, dane znajdują się pierwotnie w 12 macierzach o wymiarach 4x4 co skutkuje po przekształceniu tablicą o rozmiarze 1x192. Dane z tej tablicy zostają przekazane jako wejście do warstwy liniowej posiadającej 192 wejścia i 10 wyjść, które będą ostatnią wyjściową warstwą modelu. Schemat został ukazany na grafice 1.11



Rysunek 1.11: Schemat warstwy liniowej.

Matematycznie można zapisać operację wykonywaną w warstwie liniowej jako:

$$o_k = \sum_{j=1}^{192} (i_j * w_{j,k}) + b_k$$

gdzie:

- o_k oznacza wartość przekazaną do k-tego neurona warstwy wyjściowej,
- i oznacza wartość neurona o numerze j, z warstwy przyjmującej dane,
- $w_{j,k}$ oznacza wagę przypisaną do połączenia neuronu o numerze j z powłoki wejściowej do neurona o_k,
- b_k oznacza bias k-tego neuronu z warstwy wyjściowej, bias jest wartością stałą,
- * oznacza operację mnożenia.

Wartość największa uzyskana w warstwie wyjściowej symbolizuje etykietę cyfry, którą miał rozpoznać model.

1.5 Schemat szyfrowania homomorficznego BFV

W algebrze pojęcie homomorfizmu oznacza funkcję odwzorowującą jedną algebrę ogólną (np. monoid, grupę, pierścień czy przestrzeń wektorową) w drugą, zachowującą przy tym odpowiadające sobie działania, jakie są zdefiniowane w obu algebrach. Dzięki tej własności, przy szyfrowaniu homomorficznym, można wykonywać zdefiniowane uprzednio działania, na danych które są już zaszyfrowane. Niweluje to konieczność znajomości wartości danych, w celu wykonania obliczeń. Dzięki temu dane zachowują swoją prywatność.

Omawiany schemat szyfrowania, opiera się na problemie obliczeniowym nazwanym: "uczeniem z błędami na pierścieniach"('learning with errors over rings'). Dane reprezentowane są w postaci wielomianów. Dla danej nieujemnej liczby całkowitej n, wielomianem stopnia n-tego zmiennej x nazywane jest wyrażenie w postaci:

$$a_n x^n + a_{n-1} x^{n=1} + ... + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i$$

gdzie

$$a_0, a_1, ..., a_n$$

są współczynnikami wielomianu, należącymi do zbioru liczb całkowitych, oraz

$$a_n \neq 0$$

przykładem wielomianu jest wyrażenie:

$$x^4 + 2x^3 + x + 2$$

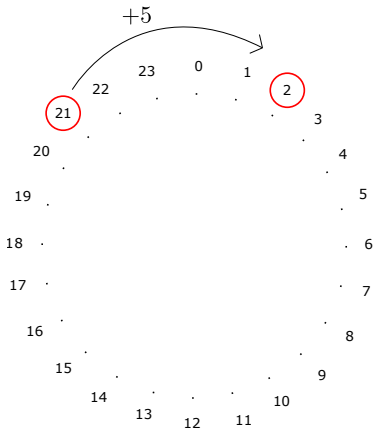
Modulo jest operacją wyznaczania reszty z dzielenia, współczynniki wielomianów w tym schemacie, należą do zbioru reszt z dzielenia przez określoną przy szyfrowaniu liczbę. Tak więc reszta z dzielenia współczynnika wielomianu n-tego stopnia wynosi:

$$a_i \mod t = r_i \quad 0 \leqslant i \leqslant n$$

gdzie t jest liczbą zdefiniowaną przy szyfrowaniu, przez którą wykonumey operację modulo. Matematycznie w szyfrowanej postaci współczynniki wielomianu są równoważne reszcie z dzielenia przez t:

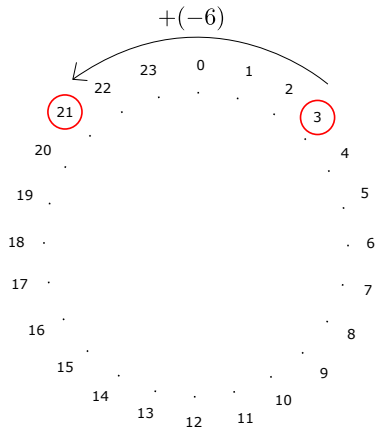
$$r_i \equiv a_i \quad (\mod n) \quad 0 \leqslant i \leqslant n$$

przykładową operację dodawania modulo t, gdzie $t = 24$ zwizualizowano na grafice 1.12 Jest to przykład dla działania $(21 + 5 = 26) \mod 24$



Rysunek 1.12: Schemat operacji modulo t dla $t = 24$ i liczb dodatnich.

Analogicznie sytuacja wygląda dla liczb ujemnych przy operacjah modulo. Na grafice 1.13 zwizualizowano przykład działania $(3 + (-6)) \mod 24$.



Rysunek 1.13: Schemat operacji modulo t dla $t = 24$ i liczb ujemnych.

Wynik działania wynosi -3 dla zbioru liczb całkowitych, jednak dla operacji na omawianym zbiorze

$$-3 \mod 24 = 21$$

w tym toku rozumowania działanie

$$(3 + (-6)) \mod 24$$

będzie równoważne z działaniem

$$(3 + 18) \mod 24$$

ponieważ

$$(-6) \mod 24 = 18$$

W schemacie szyfrowania BVF, również wielomiany są poddawane operacji modulo. Aby było to możliwe podczas szyfrowania zostaje zdefiniowany wielomian, który przyjmuje postać:

$$x^d + 1$$

gdzie $d = 2^n$ dla zdefiniowanego n. Dla przykładu niech $n = 4$. Wtedy wielomian przez który wykonywane będzie dzielenie, przyjmie postać:

$$x^{16} + 1$$

wynik dzielenia wielomianu x^{16} przez wielomian $x^{16} + 1$ przedstawiono poniżej

$$\begin{array}{r|l} & 1 \\ \hline x^{16} + 1 & x^{16} \\ & -x^{16} - 1 \\ \hline & -1 \end{array}$$

otrzymaną resztą jest -1, jednak nalezy jeszcze uwzględnić operację modulo na współczynnikach wielomianu, zatem korzystając z wartości $t = 24$ zdefiniowanej wcześniej, reszta z tego dzielenia będzie miała postać:

$$(-1) \mod 24 = 23$$

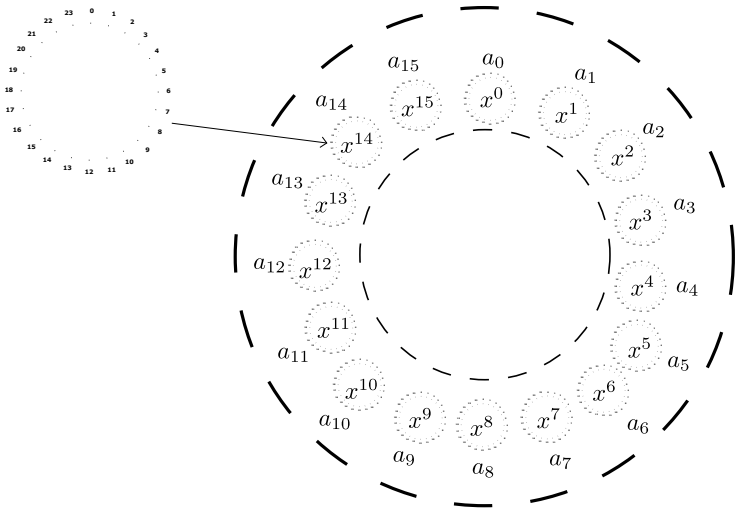
wynika z tego, że wielomiany które będą rozważane w tym przykładzie przyjmują formę:

$$a_{15}x^{15} + a_{14}x^{14} + a_{13}x^{13} + ... + a_2x^2 + a_1x + a_0$$

a każdy ze współczynników

$$a_i \quad 0 \leqslant i \leqslant 15$$

przyjmuje wartości od 0 do $t - 1$. Zapis taki można zobrazować za pomocą torusa 1.14



Rysunek 1.14: Wielomian wraz ze współczynnikami można określić na schemacie torusa.

Przy zdefiniowanej w ten sposób dziedzinie, zostanie wykonane przykładowe mnożenie wielomianów:

$$2x^{14} * x^4$$

wynikiem tego działania na zbiorze liczb całkowitych jest:

$$2x^{18}$$

dla zdefiniowanej wcześniej dziedziny zachodzi jednak operacja $\mod x^{16}+1$. Reszta z uzyskanego dzielenia wielomianów wynosi:

$$\begin{array}{r|l} & 2x^2 \\ \hline x^{16}+1 & 2x^{18} \\ & -2x^{18}-2x^2 \\ \hline & -2x^2 \end{array}$$

Jednak współczynnik wielomianu $-2x^2$ należy poddać operacji modulo dla zdefiniowanego uprzednio $t = 24$, tak więc reszta z dzielenia wielomianów będzie miała ostatecznie postać:

$$22x^2$$

Dotychczasowa część podrozdziału pokazała własności towarzyszące operacjom na pierścieniach wielomianów, na tej podstawie opiera się dalsza część, czyli sam proces szyfrowania oraz deszyfrowania danych. Proces szyfrowania odbywa się za pomocą klucza publicznego, który jest stworzony przy pomocy klucza prywatnego. Dzięki temu odszyfrowanie danych będzie możliwe tylko przy znajomości klucza prywatnego użytkownika. Dane, które będa poddawane szyfrowaniu będą zapisane w formie wielomianów należących do zdefiniowanego uprzednio pierścienia wielomianowego, oznacza to, że na wielomianie wykonane zostanie działanie:

$$\mod x^d + 1$$

gdzie $d = 2^n$, a na jego współczynnikach operacja:

$$\mod t$$

dane szyfrowane, będą reprezentowane w postaci dwóch wielomianów z tego pierścienia, jednak ich współczynniki będą poddane operacji:

$$\mod q$$

przy czym q jest dużo większe od t. Parametry używane w faktycznym kodowaniu są duże, co wpływa na bezpieczeństwo danych, jednak dla prostoty zobrazowania mechanizmu który zachodzi w tym schemacie szyfrowania, użyte zostaną małe wartości. Niech

$$d = 16 \quad t = 7 \quad q = 874$$

dla klucza prywatnego, który w przykładzie oznaczony będzie jako s, generowany jest losowo wieomian o współczynnikach ze zbioru $[-1, 0, 1]$.

$$s = x^{15} - x^{13} - x^{12} - x^{11} - x^9 + x^8 + x^6 - x^4 + x^2 + x - 1$$

następnie generowany jest wielomian dla klucza publicznego w przykładzie oznaczany jako a. Generowanie następuje również w sposób losowy, a jego współczynniki poddane są operacji $\mod q$, oczywiście zapis ujemnych współczynników występuje, ale oznaczają one wtedy liczby z pierścienia modulo.

$$\mathrm{a} = 42\mathrm{x}^{15} - 256x^{14} - 393x^{13} - 229x^{12} + 447x^{11} - 369x^{10} - 212x^9 + 107x^8 + 52x^7 + 70x^6 - 138x^5 + 322x^4 + 186x^3 - 282x^2 - 60x + 84$$

Tworzone jest również jednorazowe zaszumienie wielomianu oznaczane jako e, posiada on współczynniki o małej wartości uzyskane za pomocą dyskretnego rozkładu Gaussa oznaczanego jako:

$$\mathcal{N}(\mu, \sigma^2)$$

W standardzie szyfrowania homomorficznego [11], wartości (μ, σ) zostały zdefiniowane jako $(0, \frac{8}{\sqrt{2\pi}} \approx 3.2)$

$$e = -3x^{15} + x^{14} + x^{13} + 7x^{12} - 6x^{11} - 6x^{10} + x^9 + 4x^8 - x^6 + 3x^5 - 4x^4 + 4x^3 + 4x + 1$$

Klucz publiczny zostaje zdefiniowany jako para wielomianów:

$$p_k = ([-as + e]_q, a)$$

gdzie

$$[wielomian]_q$$

oznacza operację $\mod q$ na współczynnikach wielomianu, należy pamiętać, że sam wielomian również zostaje poddany operacji $\mod x^d + 1$. Po wykonaniu działań pierwszy element klucza publicznego przyjmie postać:

$$\mathrm{p}_{k_0} = -285x^{15} - 431x^{14} - 32x^{13} + 86x^{12} - 83x^{11} - 142x^{10} - 41x^9 + 430x^8 + 26x^7 - 158x^6 - 281x^5 + 377x^4 + 110x^3 - 234x^2 - 113x + 252$$

Aby złamać szyfrowanie należałoby poznać klucz prywatny s, jeżeli nie zostałoby użyte zaszumienie byłoby to względnie proste, ze względu na występowanie w kluczu publicznym wielomianu a. Mimo zaszumienia wielomianem e dla podanego przykładu, złamanie kodu byłoby możliwe za pomocą metody 'brute force', która sprawdzałaby każdy możliwy wielomian s z wyrażenia $-as + e$, ponieważ współczynniki klucza prywatnego należą do zbioru $[-1, 0, 1]$, a potęgi wielomianu do zbioru $[0, 1, 2, ..., 14, 15]$ skutkuje to łączną liczbą $3^{16} = 43046721$ prób. Szukana wartość ze względu na występujący szum, nie będzie równa wielomianowi a, nie mniej najbardziej zbliżony wielomian uzyskany metodą 'brute force' oznacza znalezienie klucza prywatnego. Dlatego w schemacie szyfrowania używa się dużych liczb, przykładowo, dla $d = 4096$ ilość przypadków do sprawdzenia wzrosłaby do 3^{4096}, co staje się niewykonalnym dla współczesnych komputerów.

Szyfrowanie danych przypomina proces tworzenia klucza publicznego. Niech wiadomość, która ma być zaszyfrowana będzie oznaczana w przykładzie jako $m = 3+4x^8$. Jest ona w postaci wielomianu po operacji mod $x^d + 1$, a na jego współczynnikach wykonana jest operacja mod t. Na tym pierścieniu zachodzi równoważność wyrażeń $3 + 4x^8 \equiv 3 - 3x^8$. Do zaszyfrowania zostają jednorazowo stworzone dodatkowe trzy wielomiany. Dwa pełniące funkcję zaszumienia w przykładzie oznaczane kolejno jako e_1, e_2, które jak w przypadku szumu klucza publicznego posiadają współczynniki o małej wartości uzyskane za pomocą dyskretnego rozkładu Gaussa, oraz najwyższy możliwy stopień ich wielomianu to $d - 1$.

$$\mathrm{e}_1 = -5x^{15} - 2x^{14} - 3x^{13} - x^{12} - 4x^{11} + 3x^{10} + x^9 + 4x^8 + 4x^7 + 5x^6$$
$$\text{- } 4\mathrm{x}^5 - 3x^4 - 3x^3 + 2x^2 - 6x + 4$$
$$\mathrm{e}_2 = -7x^{15} + 2x^{14} - 4x^{13} + 5x^{11} + 2x^{10} - x^9 + 4x^8 - 4x^7 - 3x^6 + 2x^5 -$$
$$2\mathrm{x}^4 + x^3 - 4x^2 - 2x + 2$$

trzeci wielomian oznaczany w przykładzie jako u, posiada współczynniki ze zbioru $[-1, 0, 1]$, które tak jak w przypadku klucza prywatnego, są przydzielane w sposób losowy, oraz najwyższy możliwy stopień wielomianu u to $d - 1$.

$$u = x^{14} + x^{13} + x^{12} - x^8 - x^5 - x^3 + 1$$

tekst szyfrowany oznaczany w przykładzie jako c_t będzie miał postać:

$$c_t = ([p_{k_0}u + e_1 + \frac{q}{t}m]_q, [p_{k_1}u + e_2]_q)$$

współczynniki wiadomości m zostają przeskalowane przez wartość $\frac{q}{t}$, która w tym przykładzie będzie równa 128, służy to zamaskowaniu zakresu operacji mod q, której poddawane są wyrażenia c_{t_0} oraz c_{t_1}. Wyrażenia $p_{k_0}u + e_1$ służą zakryciu wiadomości. Losowy sposób dobierania współczynników wielomianu u sprawia, że ta sama wiadomość będzie skutkowała inną zaszyfrowana postacią za każdym razem, ma to na celu sprawić aby niemożliwym było zdefiniowanie, że zaszyfrowana postać dotyczy tych samych danych. Podstawiając $p_{k_0} = [-as + e]_q$, można zapisać c_{t_0} jako:

$$c_{t_0} = [e_1 + eu - aus + \frac{q}{t}m]_q$$

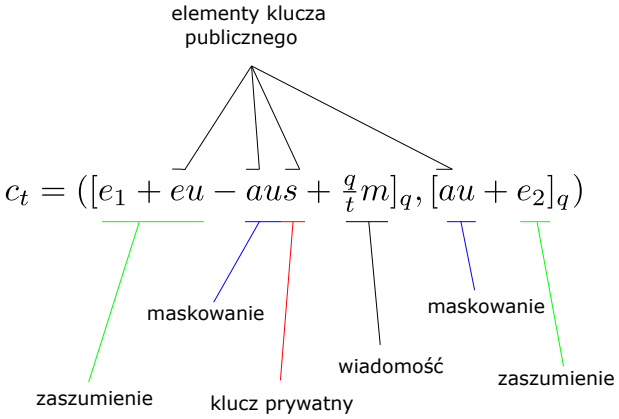
pierwsze dwa elementy tego wyrażenia są małe pod względem współczynników przy wielomianie, pozostałe dwa są duże, a pierwszy z nich maskuje wiadomość. Podstawiając $p_{k_1} = a$, można zapisać c_{t_1} jako:

$$c_{t_1} = [au + e_2]_q$$

po rozpisaniu działań elementy c_t przyjmą postać:

$$c_{t_0} = 217x^{15} - 53x^{14} + 13x^{13} - 249x^{12} - 392x^{11} - 238x^{10} + 252x^9 + 115x^8 + 5x^7 + 184x^6 - 201x^5 - 258x^4 - 247x^3 + 144x^2 + 23x + 42$$
$$c_{t_1} = 25x^{15} + 225x^{14} - 12x^{13} + 270x^{12} + 350x^{11} - 24x^{10} + 56x^9 - 330x^8 + 386x^7 + 225x^6 - 332x^5 + 68x^4 - 20x^3 - 26x^2 - 91x + 380$$

podsumowując proces szyfrowania wiadomości, przedstawiona została grafika 1.15 wymieniająca elementy składające się na wiadomość po zakodowaniu.



Rysunek 1.15: Zaszyfrowana wiadomość

przy odszyfrowywaniu wiadomości niezbędne jest użycie klucza prywatnego s. Wyrażenie $c_{t_1} = [au + e_2]_q$ zostaje za jego pomocą przekształcone w:

$$c_{t_1}s = [aus + e_2 s]_q$$

posłuży to do pozbycia się wyrażeń maskujących wiadomość z c_{t_0}. Proces wygląda w sposób następujący:

$$c_{t_0} + c_{t_1}s = [e_1 + eu - aus + \frac{q}{t}m + aus + e_2 s]_q = [\frac{q}{t}m + eu + e_1 + e_2 s]_q$$

na składniki, które pozostały składają się przeskalowana wiadomość oraz elementy zaszumienia, które posiadają małe współczynniki. Proces odzyskania wiadomości będzie polegał na przeskalowaniu wyrażenia przez $\frac{t}{q}$, a następnie zaokrągleniu wartości współczynników do najbliższych im liczb całkowitych. Po przeskalowaniu współczynniki wielomianu ponownie znajdą się w pierscieniu $\mod t$ Po podstawieniu wartości liczbowych uzyskany zostanie wynik:

$$c_{t_1}s + c_{t_0} = 13x^{15} - 2x^{14} + 17x^{13} + 22x^{12} - 32x^{11} - 23x^{10} + 19x^9 - 30x^8 + 9x^7 + 10x^6 - 13x^5 - 3x^4 - 2x^3 - 12x^2 + 7x + 393$$

$$(c_{t_1}s + c_{t_0})\tfrac{7}{874} = \tfrac{13}{128}x^{15} - \tfrac{1}{64}x^{14} + \tfrac{17}{128}x^{13} + \tfrac{11}{64}x^{12} - \tfrac{1}{4}x^{11} - \tfrac{23}{128}x^{10} + \tfrac{19}{128}x^9 - \tfrac{95}{32}x^8 + \tfrac{9}{128}x^7 + \tfrac{5}{64}x^6 - \tfrac{13}{12}x^5 - \tfrac{3}{128}x^4 - \tfrac{1}{64}x^3 - \tfrac{3}{32}x^2 + -\tfrac{1}{128}x + \tfrac{393}{128}$$

$$\lfloor (c_{t_0} + c_{t_1}s)\frac{7}{874} \rceil = 3 - 3x^8 = m$$

formułę odszyfrowywania można zatem zapisać jako:

$$[\lfloor (c_{t_0} + c_{t_1}s)\frac{t}{q} \rceil]_t$$

należy zaznaczyć, że w przypadku zbyt wysokiego zaszumienia przybliżenie współczynników w etapie końcowym odszyfrowywania skończyłoby się błędną wiadomością zwrotną. Kluczowym jest zastosowanie względnie niskich wartości przy współczynnikach szum oraz odpowiedniego stosunku $\frac{q}{t}$

1.5.1 Dodawanie zaszyfrowanych danych

Matematyczne operacje, którym poddawane są zaszyfrowane uprzednio dane to homomorficzne dodawanie oraz mnożenie. Umożliwiają one działania na danych bez konieczności ich odszyfrowania. Pierwszym rozptrywanym przypadkiem jest dodawanie, niech m_1, m_2 oznaczają dwie wiadomości, które zostaną poddane szyfrowaniu, bazując na dotychczasowych wzorach, przy pomocy tego samego klucza publicznego uzyskane zostaną zaszyfrowane wyniki w postaci:

$$a = ([p_{k_0}u_1 + e_1 + \frac{q}{t}m_1]_q, [p_{k_1}u_1 + e_2]_q)$$

$$b = ([p_{k_0}u_2 + e_3 + \frac{q}{t}m_2]_q, [p_{k_1}u_2 + e_4]_q)$$

wiadomości zaszyfrowane różnią się poza wartościami m_1, m_2 innym zaszumieniem oraz wielomianami u_1, u_2. Poniżej rozpisano dodawanie homomorficzne dla a i b.

$$a + b = ([p_{k_0}(u_1 + u_2) + (e_1 + e_3) + \frac{q}{t}(m_1 + m_2)]_q, [p_{k_1}(u_1 + u_2) + (e_2 + e_4)]_q)$$

dla uproszczenia równania wprowadzone zostaną dodatkowe oznaczenia:

$$e_5 = e_1 + e_3$$

$$u_3 = u_1 + u_2$$

$$e_6 = e_2 + e_4$$

zaszyfrowany wynik działania $c = a + b$ przyjmie wtedy postać:

$$c = ([p_{k_0}(u_3) + (e_5) + \frac{q}{t}(m_1 + m_2)]_q, [p_{k_1}(u_3) + (e_6)]_q)$$

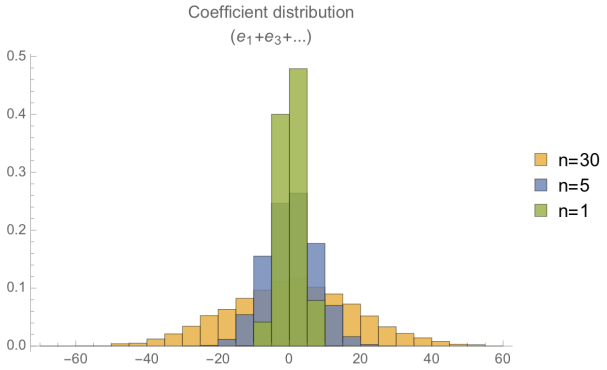
odszyfowanie odbywa się za pomocą zaprezentowanej wcześniej formuły:

$$[\lfloor (c_0 + c_{t_1}s)\frac{t}{q} \rceil]_t$$

kluczowym punktem jest działanie:

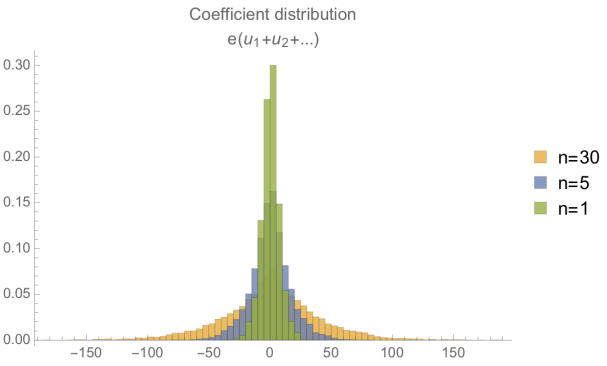
$$c_0 + c_{t_1} s = [\frac{q}{t}(m_1 + m_2) + e_5 + eu_3 + e_6 s]_q$$

ponieważ w przypadku zbyt wysokiego zaszumienia (wartości większe niz $\frac{q}{2t}$) po przeskalowaniu przez $\frac{t}{q}$, a następnie zaokrągleniu do najbliższej liczby całkowitej, uzyskana wiadomość będzie błędna. Również w miarę zwiększania się ilości działań dodawania i mnożenia na zaszyfrowanych danych, zwiększa się ryzyko uzyskania zbyt wysokich współczynników przy wielomianach zaszumienia. Dla zwiualizowania tego problemu posłużono się badaniami uzyskanymi w artykule [9]. W przypadku dodawania zaszumień:1.16



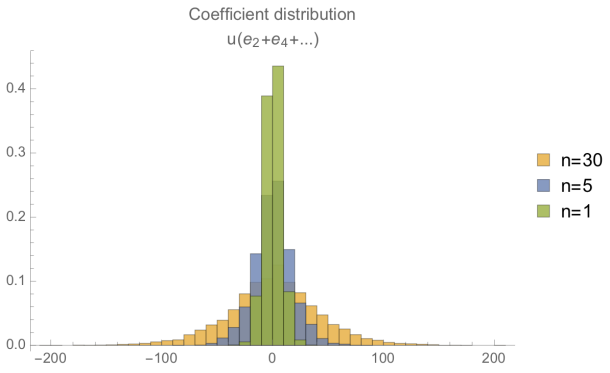
Rysunek 1.16: Rozkład prawdopodobieństwa od wartości współczynników po operacjach matematycznych, przez n oznaczona została ilość działań.

W przypadku mnożenia zaszumienia z wielomianami maskującymi u:1.17



Rysunek 1.17: Rozkład prawdopodobieństwa od wartości współczynników po operacjach matematycznych, przez n oznaczona została ilość działań.

W przypadku mnożenia zaszumień z wielomianem maskującym u:1.18



Rysunek 1.18: Rozkład prawdopodobieństwa od wartości współczynników po operacjach matematycznych, przez n oznaczona została ilość działań.

1.5.2 Mnożenie zaszyfrowanych danych

Drugim rozpatrywanym przypadkiem jest operacja mnożenia homomorficznego. Proces odszyfrowywania wygląda nieco inaczej niż w przypadku dodawania. Poniżej przedstawiono proces usuwania maski z przykładowej zaszyfrowanej wiadomości c_t, gdzie klucz prywatny oznaczamy jako s, ten proces jest analogiczny do procesu w przypadku dodawania.

$$[c_{t_0}s^0 + c_{t_1}s^1]_q$$

Można przedstawić ten proces w formie wielomianu z potęgami s, gdzie współczynnikami są c_{t_0}, c_{t_1}. Upraszczając zapis otrzymany zostanie wynik:

$$[c_{t_0}s^0 + c_{t_1}s^1]_q \rightarrow \frac{q}{t}m + noise$$

korzystając z powyżej rozpisanych własności, niech a, b oznaczają zaszyfrowane wiadomości m_1, m_2:

$$[a_0s^0 + a_1s^1]_q \rightarrow \frac{q}{t}m_1 + n_1$$

$$[b_0s^0 + b_1s^1]_q \rightarrow \frac{q}{t}m_2 + n_2$$

w powyższym zapisie n_1, n_2 symbolizują zaszumienie dla kazdej z wiadomości. Mnożenie tych wiadomości można więc przedstawić jako:

$$[a_0s^0 + a_1s^1]_q[b_0s^0 + b_1s^1]_q \rightarrow (\frac{q}{t}m_1 + n_1)(\frac{q}{t}m_2 + n_2)$$

wyrażenie po prawej stronie jest niezależne od wielomianów, które służyły maskowaniu wiadomości, z tego wynika, że to samo dotyczy wyrażenia po stronie lewej. Wprowadzając nowe oznaczenia, oraz dokonując dodatkowego przeskalowania lewej strony przez $\frac{t}{q}$, wynikającego z powodu otrzymania po mnożeniu wyrażenia $\frac{q^2}{t^2}m_1m_2$ można ją zapisać jako:

$$c_0 + c_1s + c_2s^2$$

gdzie:

$$c_0 = [\frac{t}{q}a_0b_0]_q$$

$$c_1 = [\frac{t}{q}(a_1b_0 + a_0b_1)]_q$$

$$c_2 = [\frac{t}{q}a_1 b_1]_q$$

uogólniając formułe deszyfrowania otrzymany zostaje zapis:

$$[\lfloor \frac{t}{q}[c_0 s^0 + c_1 s^1 + c_2 s^2]_q \rceil]_t$$

poniżej rozpisano przebieg powyższej formuły:

$$p_k = ([-as + e]_q, a)$$

$$a = [[p_{k_0}u_1 + e_{1,1} + \frac{q}{t}m_1]_q, [p_{k_1}u_1 + e_{1,2}]_q]$$

$$b = [[p_{k_0}u_2 + e_{2,1} + \frac{q}{t}m_2]_q, [p_{k_1}u_2 + e_{2,2}]_q]$$

$$\begin{gathered}
\mathrm{c}_0 + c_1 s + c_2 s^2 \\
= \\
\tfrac{t}{q}(p_{k_0}u_1 + e_{1,1} + \tfrac{q}{t}m_1)(p_{k_0}u_2 + e_{2,1} + \tfrac{q}{t}m_2) \\
+ \\
\tfrac{t}{q}((p_{k_1}u_1 + e_{1,2})(p_{k_0}u_2 + e_{2,1} + \tfrac{q}{t}m_2) \\
+ \\
(p_{k_0}u_1 + e_{1,1} + \tfrac{q}{t}m_1)(p_{k_1}u_2 + e_{2,2}))s \\
+ \\
\tfrac{t}{q}(p_{k_1}u_1 + e_{1,2})(p_{k_1}u_2 + e_{2,2})s^2 \\
= \\
\tfrac{t}{q}((-as + e)u_1 + e_{1,1} + \tfrac{q}{t}m_1)((-as + e)u_2 + e_{2,1} + \tfrac{q}{t}m_2) \\
+ \\
\tfrac{t}{q}((au_1 + e_{1,2})((-as + e)u_2 + e_{2,1} + \tfrac{q}{t}m_2) \\
+ \\
((-as + e)u_1 + e_{1,1} + \tfrac{q}{t}m_1)(au_2 + e_{2,2}))s \\
+ \\
\tfrac{t}{q}(au_1 + e_{1,2})(au_2 + e_{2,2})s^2 \\
= \\
\tfrac{q}{t}m_1 m_2 + e_{2,2}m_1 s + e_{1,2}m_2 s + em_2 u_1 + em_1 u_2 + e_{2,1}m_1 + e_{1,1}m_2 \\
+ \\
\tfrac{t}{q}e^2 u_1 u_2 + \tfrac{t}{q}e_{1,2}e_{2,2}s^2 + \tfrac{t}{q}e_{2,2}esu_1 + \tfrac{t}{q}e_{1,2}esu_2 \\
+ \\
\tfrac{t}{q}e_{1,2}e_{2,1}s + \tfrac{t}{q}e_{1,2}e_{2,1}s + \tfrac{t}{q}e_{1,1}e_{2,2}s + \tfrac{t}{q}e_{2,1}eu_1 + \tfrac{t}{q}e_{1,1}eu_2 + \tfrac{t}{q}e_{1,1}e_{2,1}
\end{gathered}$$

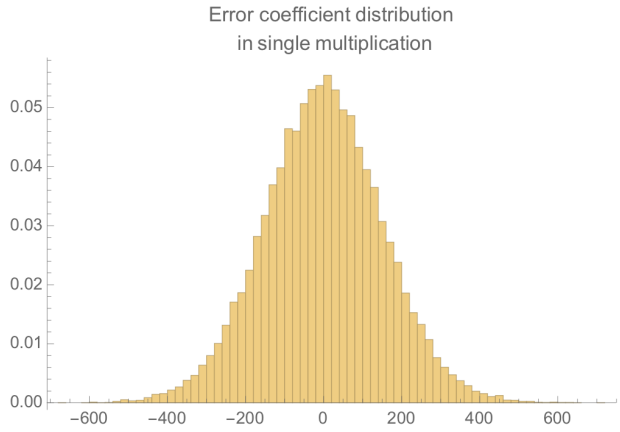
po podstawieniu uzyskanego zapisu do zapisanej wcześniej formuły deszyfrującej, uzsyka się pierwotnie zaszyfrowane przemnożone już wiadomości w postaci $m_1 m_2$. Jednak w przypadku gdy zaszumienie okazałoby się zbyt wysokie, wynik może być błędny. W artykule [9], przeprowadzono badania prawdopodobieństwa uzyskania zbyt dużych wyników zaszumienia dla parametrów szyfrowania:

$$d = 16$$

$$t = 7$$

$$q = 7168 = 1024 * t$$

dla tych parametrów błędne wyniki zostaną uzyskane w przypadku współczynników, któe osiągną wartość $\frac{q}{2t} = 512$. Po uogulnienieniu uzyskanych zaszumień do jednej postaci uzyskano wykres 1.19



Rysunek 1.19: Rozkład prawdopodobieństwa od wartości współczynników po operacji mnożenia.

Jak widać na załączonej grafice istnieje możliwość uzyskania niewłaściwego wyniku już przy jednej operacji mnożenia homomorficznego, jeżeli stosunek parametrów szyfrowania q i t jest zbyt niski.

W przykładzie poruszony został przypadek jednej operacji mnożenia, jednak w praktyce, operacji takich może być więcej, produkuje to większy rozmiar szyfrowanej wiadomości co skutkuje większym zużyciem pamięci oraz złożoności obliczeniowej. Aby temu zapobiedz w schemacie BFV po każdej operacji mnożenia stosowana jest relinearyzacja, której zadaniem jest zmniejszanie rozmiaru wiadomości szyfrowanej, dla zobrazowania wiadomość długości n zostanie zmniejszona do rozmiaru $n-1$, gdzie najmniejszy rozmiar to 2.

$$[c_{t_0}s^0 + c_{t_1}s^1]_q$$

do wykonania tej operacji używa się klucza (evaluation key), którego formułę można zapisać, zgodnie ze stosowanymi do tej pory oznaczeniami jako:

$$ev_k = ([-(as+e) + s^{n-1}]_q, a)$$

gdzie n oznacza stopień wielomianu który ma zostać zredukowany. Dla przykładu niech n = 3. Wielomian który zostanie poddany realineryzacji ma wtedy postać:

$$[c_0s^0 + c_1s^1 + c_2s^2]_q$$

zadaniem realineryzacji będzie sprowadzenie ego wielomianu do postaci:

$$[c'_0s^0 + c'_1s^1]_q$$

jest to możliwe poprzez przypisanie c'_0, c'_1 następujacych wartości:

$$c'_0 = c_0 + ev_{k_0}c_2$$

$$c'_1 = c_1 + ev_{k_1}c_2$$

dla potwierdzenia zapisu zostało wyprowadzone równanie:

$$c'_0s^0 + c'_1s^1 = c_0 + ev_{k_0}c_2 + s(c_1 + ev_{k_1}c_2) = c_0 + c_1s + c_2(ev_{k_0} + ev_{k_1}s) = c_0 + c_1s + c_2s^2 + c_2e$$

po wyprowadzeniu równania pozostał jednak dodatkowy element c_2e, który w praktyce może wpływać na zakłamanie rzeczywistego wyniku, proces zminimalizowania tego błędu został opisany w artykule [8], w niniejszej pracy proces ten nie został uwzględniony ponieważ celem jej, jest jedynie zarysować schemat szyfrowania BFV.

Rozdział 2

Implementacja systemu

W niniejszym rozdziale przedstawiono, w jaki sposób modele sieci neuronowych opisanych w rozdziale pierwszym poddawane są operacji szyfrowania homomorficznego. Przedstawiony został język programowania, biblioteki językowe oraz środowisko w którym projekt został zaimplementowany. Omawiane procesy zawierają kluczowe fragmenty kodu, grafiki pomocnicze oraz szczegółowe opisy przebiegu.

2.1 Przegląd użytych technologii

Cały kod źródłowy został stworzony w chmurze na platformie Google Colaboratory [1]. Językiem pro- gramowania używanym na platformie Google Colaboratory jest język Python [4] w wersji obsługiwanej w chmurze środowiskowej. Do modelowania sieci neuronowych oraz procesu ich nauczania wykorzystano bibliotekę PyTorch [5] w najnowszej dostępnej aktualnie wersji na platformie chmurowej. Dla zwizualizowania procesów zachodzących podczas treningów oraz testów modeli posłużono się biblioteką Tensorboard [6]. Celem ułatwienia implementacji przeprowadzanych operacji matematycznych posłużyła biblioteka NumPy [2]. Za źródło danych wejściowych dla sieci neuronowej, posłużono się zbiorem danych MNIST [7]. Biblioteką, która umożliwia zaszyfrowanie zaimplementowanych modeli sieci schematem BFV jest Pyfhel [3].

2.2 Pobieranie danych treningowych i testowych

Do trenowania i testowania obydwu modeli sieci posłużą dane ze zbioru MNIST. Są to obrazy pisma odręcznego o wymiarach 28 pikseli na 28 pikseli. Biblioteka PyTorch dostarcza funkcjonalność pozwalającą na pobranie tych danych. Implementacja została przedstawiona w kodzi źródłowym 2.1

```
# zaimportowanie niezbędnych bibliotek
import torch
import torchvision

# przypisanie funkcji konwertującej dane o obrazie
# z tablicy zawierającej wartości w zakresie [0, 255]
# do tablicy zawierającej wartości w zakresie [0, 1.0]
transform = transforms.ToTensor()

# pobranie danych treningowych
# root: katalog pobierania
# train: wartość informująca czy dane są treningowe czy testowe
# transform: zawiera informacje o sposobie konwertowania danych
train_set = torchvision.datasets.MNIST(
    root = './data',
    train=True,
    download=True,
    transform=transform
)

```

```
# pobranie danych testowych
test_set = torchvision.datasets.MNIST(
    root = './data',
    train=False,
    download=True,
    transform=transform
)

# zainicjowanie iteracji przez pobrane dane
# z określoną ilością ładowanych danych
# train_set: oznacza zbiór który zostanie załadowany
# batch_size: ilość danych załadowanych w iteracji pobierania z całego zbioru
# shuffle: wartość decydująca o przemieszaniu danych
train_loader = torch.utils.data.DataLoader(
    train_set,
    batch_size=50,
    shuffle=True
)

# zainicjowanie iteracji przez pobrane dane
# z określoną ilością ładowanych danych
test_loader = torch.utils.data.DataLoader(
    test_set,
    batch_size=50,
    shuffle=True
))
```

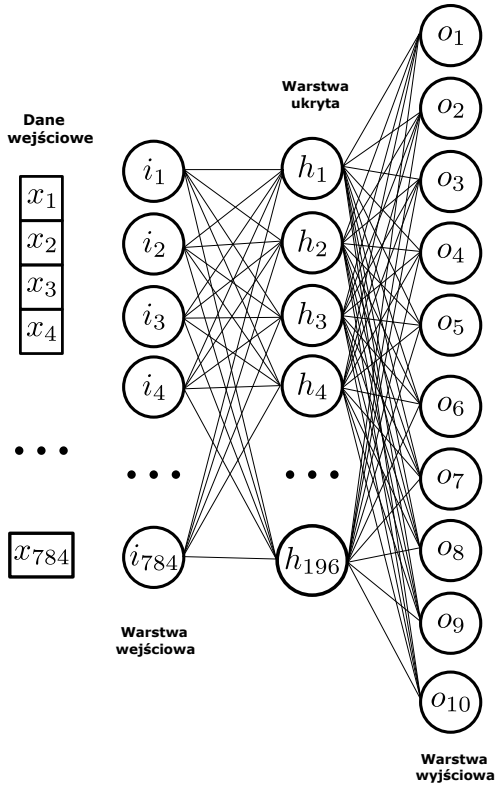
Kod źródłowy 2.1: Skrypty odpowiedzialne za pobieranie danych ze zbioru MNIST.

Kod źródłowy 2.1 przedstawia skrypty służące do pobrania danych treningowych i danych testowych ze zbioru MNIST. W skryptach określone są: wybór zbioru treningowego lub testowego, liczba obrazów, które zostaną załadowane podczas pobierania danych ze zbioru, skalowanie wartości liczbowych reprezentujących kolor obrazu, przemieszanie danych).

2.3 Implementacja modeli sieci.

2.3.1 Model liniowy.

W rozdziale pierwszym przybliżone zostały dwa modele sieci neuronowych. Pierwszym z nich jest model liniowy 2.1 posiadający warstwę wejściową, jedną warstwę ukrytą oraz warstwę wyjściową.



Rysunek 2.1: Schemat pierwszego modelu sieci

Kod źródłowy w którym następuje zainicjowanie sieci liniowej przedstawiono na 2.2

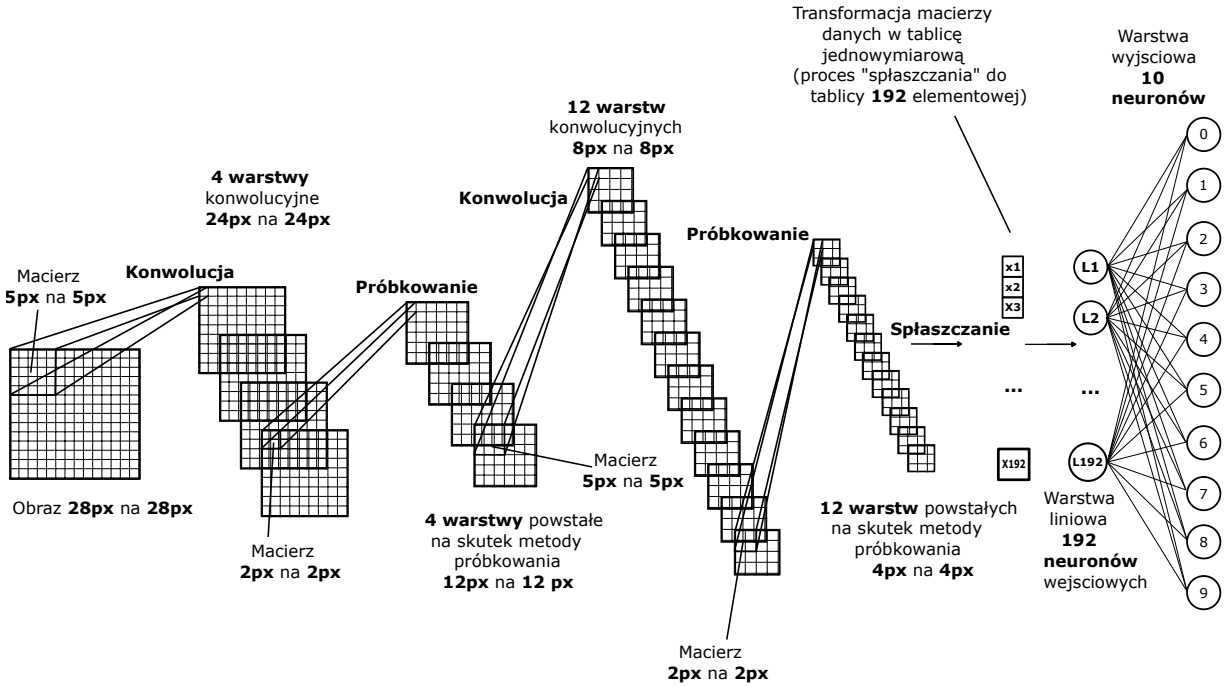
```
# zaimportowanie niezbędnych bibliotek
import torch.nn as nn

# inicjowanie warstw modelu w odpowiedniej kolejności
Lin_Net1 = nn.Sequential(
        # warstwa spłaszczająca dane
        nn.Flatten(),

        # warstwa liniowa
        nn.Linear(784, 196),

        # funkcja aktywacyjna przypisana do kolejnej warstwy
        nn.Tanh(),

        # warstwa liniowa
        nn.Linear(196, 10),
    )
```

Kod źródłowy 2.2: Zainicjowanie modelu liniowej sieci neuronowej

Kod źródłowy 2.2 przedstawia zainicjowanie obiektu klasy reprezentującej sieć liniową. Zaimplementowane zostały: warstwa spłaszczająca dane wejsciowe, warstwa liniowa posiadająca 784 wejścia i 196 wyjść, funkcja aktywacyjna którą jest tanges hiperboliczny, warstwa ukryta posiadająca 196 wejść i 10 wyjść.

2.3.2 Model LeNet1

Drugi model sieci to opisany w rozdziale pierwszym model LeNet1, składający się z warstw konwolucyjnych, próbkujących oraz liniowej warswty wyjściowej 2.2.



Rysunek 2.2: Schemat modelu sieci LeNet1

Kod źródłowy w którym następuje zainicjowanie tej sieci przedstawiono na 2.3

```
# zaimportowanie niezbędnych bibliotek
import torch.nn as nn

# inicjowanie warstw modelu w odpowiedniej kolejności
LeNet1 = nn.Sequential(
        # warstwa konwolucyjna
        nn.Conv2d(1, 4, kernel_size=5),

        # funkcja aktywacyjna
        nn.Tanh(),

        # warstwa próbkująca
        nn.AvgPool2d(kernel_size=2),

        # warstwa konwolucyjna
        nn.Conv2d(4, 12, kernel_size=5),

        #funkcja aktywacyjna
        nn.Tanh(),

        # warstwa próbkująca
        nn.AvgPool2d(kernel_size=2),

        # warstwa spłaszczająca dane
        nn.Flatten(),

        # warstwa liniowa
        nn.Linear(192, 10),
    )
```

Kod źródłowy 2.3: Zainicjowanie modelu LeNet1

Kod źródłowy 2.3 przedstawia zainicjowanie obiektu klasy reprezentującej sieć LeNet1. Zaimplementowane zostały: warstwa konwolucyjna posiadająca 4 mapy funkcji z macierzą filtra mapującego o wymiarach

5 na 5, funkcją aktywacyjna, którą jest tanges hiperboliczny, warstwa próbkująca posiadająca macierz filtra próbkującego o wymirach 2 na 2, warstwa konwolucyjna posiadająca 3 mapy funkcji z macierzą filtra mapującego o wymiarach 5 na 5, funkcją aktywacyjna, którą jest tanges hiperboliczny, warstwa próbkująca posiadająca macierz filtra próbkującego o wymirach 2 na 2, warstwa spłaszczająca dane wejsciowe, warstwa liniowa posiadająca 192 wejścia i 10 wyjść.

2.3.3 Zmiana funkcji aktywacyjnej

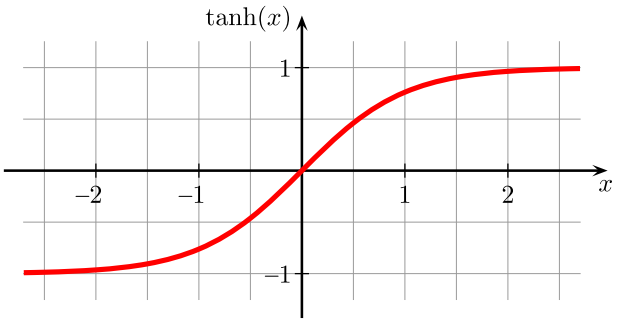
Należy zaznaczyć, że operacje homomorficzne, które zostały zdefiniowane w schemacie BFV to dodawanie i mnożenie. Pozwala to na przedstawianie liniowych funkcji. Zgodnie z definicją przekształcenie liniowe musi spełniać własności:

$$f(x+y) = f(x) + f(y)$$

Oraz przy mnożeniu przez wartość skalującą c (skalar):

$$f(cx) = cf(x)$$

Jednak w modelach zdefiniowanych w pracy występuje funkcja aktywacyjna, którą jest tanges hiperboliczny 2.3.



Rysunek 2.3: Tanges hiperboliczny

Matematyczna formuła wygląda w sposób następujący:

$$tanh(x) = \frac{e^{2x}+1}{e^{2x}-1}$$

Gdzie e jest liczbą eulera wyrażaną jako:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Z zasobem dostępnych operacji homomorficznych, nie uda się zaimplementować nieliniowości tej funkcji. Można jednak zastąpić tę funkcję aktywacyjną poprzez inną. W projekcie użyto w tym celu funkcji:

$$x^2$$

Która jest prostą nieliniową funkcją wykorzystującą mnożenie, a to z kolei jest możliwe do zaimplementowania w szyfrowaniu homomorficzym. W tym celu zaimplementowana została klasa *Square*() 2.4, zawierająca funkcję zwracającą wartość podanego argumentu podniesioną do potęgi drugiej. Aby klasa mogła zostać wykorzystana w modelu sieci do implementacji wykorzystana zostałą biblioteka PyTorch.

```
# zaimportowanie niezbędnych bibliotek
import torch
import torch.nn as nn

# implementacja klasy
```

```
# nn.Module jest podstawową klasą
# dla modeli sieci budowanych z biblioteki PyTorch
class Square(nn.Module):

    # konstruktor inicjalizuje obiekt klasy
    def __init__(self):

        # super() pozwala na dziedziczenie metod klasy nadrzędnej
        super().__init__()

    # metoda zwracająca wartość podanego argumentu
    # podniesioną do potęgi drugiej
    def forward(self, t):
        return torch.pow(t, 2)
```

Kod źródłowy 2.4: Implementacja klasy Square

Kod źródłowy 2.4 przedstawia klasę reprezentującą funkcję aktywacyjną x^2 Zaimplementowane zostały: konstruktor klasy, funkcja zwracająca wartość argumentu podniesioną do potęgi drugiej.

Modele ze zmienioną funkcją aktywującą będą implementowane w postaci: 2.5 oraz 2.6

```
Lin_Net1 = nn.Sequential(
    nn.Flatten(),

    nn.Linear(784, 196),

    # zmieniona funkcja aktywacyjna
    Square(),

    nn.Linear(196, 10),
    )
```

Kod źródłowy 2.5: Pierwszy model sieci z funkcją aktywacyjną x^2

Kod źródłowy 2.5 przedstawia zainicjowanie obiektu klasy reprezentującej sieć liniową z funkcją aktywacyjną x^2.

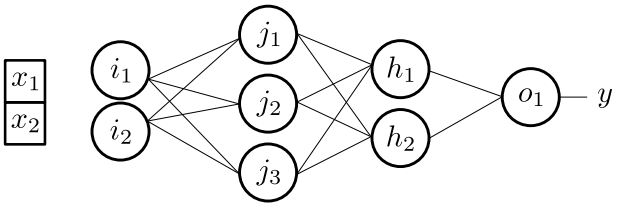
```
# zaimportowanie niezbędnych bibliotek
import torch.nn as nn

LeNet1_Approx = nn.Sequential(
    nn.Conv2d(1, 4, kernel_size=5),

    Square(),

    nn.AvgPool2d(kernel_size=2),

    nn.Conv2d(4, 12, kernel_size=5),

    # zmieniona funkcja aktywacyjna
    Square(),

    nn.AvgPool2d(kernel_size=2),

    nn.Flatten(),

    nn.Linear(192, 10),
    )
```

Kod źródłowy 2.6: Model sieci LeNet1 z funkcją aktywacyjną x^2

Kod źródłowy 2.6 przedstawia zainicjowanie obiektu klasy reprezentującej sieć LeNet1 z funkcją aktywacyjną x^2.

2.4 Trenowanie sieci

Trenowanie sieci polega na dostosowaniu wag połączeń między neuronami w modelu, w taki sposób aby sieć zwracała najbardziej prawdopodobne wartości. Aby było to możliwe wykorzystywane są algorytmy propagacji wstecznej, oraz współczynniki uczenia sieci. Dla zobrazowania mechanizmu propagacji wstecznej wykorzystany został prosty model sieci przedstawiony na grafice 2.4.

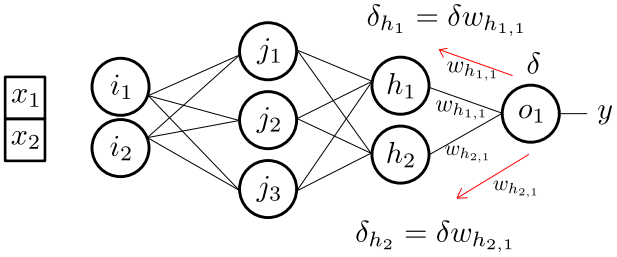


Rysunek 2.4: Sieć posiadająca 3 warstwy

Sieć podczas treningu zwraca pewną wartość y, ta z kolei porównywana jest z aktualną wartością, która przykładzie oznaczona jest jako z. Następnie wyliczana jest różnica między wartościami:

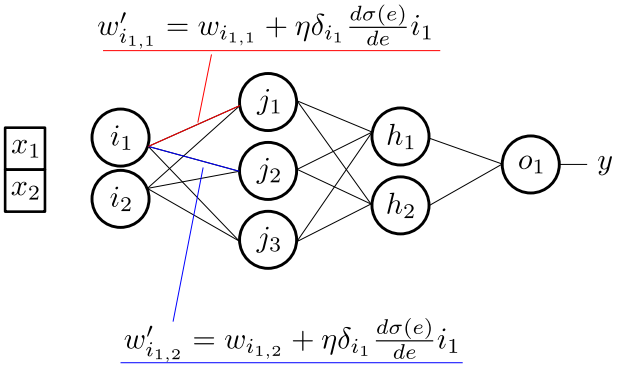
$$\delta = z - y$$

Następnie wartość błędu jest propagowana wstecz, do neuronów warstw poprzedzających warstwę bieżąca. Wykorzystywane w tym celu są wagi połączeń między neuronami. 2.5.



Rysunek 2.5: Propagacja błędu wstecz

Schemat ten jest powtarzany do momentu wyliczenia wartośći błędu dla każdego z neuronów. Następnie dostosowywane są wagi połączeń neuronów w modelu zgodnie ze wzorem przedstawionym na grafice: 2.6.



Rysunek 2.6: Dostosowanie wag na bazie błędu uzyskanego z propagacji wstecznej

Gdzie:

- $w'_{i1,1}$ oznacza nową wartość wagi,
- $w_{i1,1}$ oznacza dotychczasową wartość wagi,
- η oznacza współczynnik uczenia sieci, który decyduje jak bardzo waga powinna się zmienić, wraz ze wzrostem dokładności sieci, współczynnik ten maleje,
- δ_{i_1} wartość błędu przypisana przy neuronie i_1 uzyskana ze wstecznej propagacji,
- $\sigma(e)$ oznacza funkcję aktywacyjną, której argumentem jest e, w przypadku neurona i_1 wartość $e = x_1$, jednak dla uogólnienia wzoru zapisana została wartość e,
- $\frac{d}{de}$ oznacza pochodną funckji aktywacyjnej po argumencie jaki przyjmuje e,
- i_1 oznacza wartość przechowywaną przez neuron i_1 .

Proces ten wykonywany jest dla wszystkich wag modelu sieci, przy każdej iteracji treningowej. Wraz z kolejnymi krokami wykonywanymi podczas treningu, wagi dostosowują model do coraz dokładniejszych predykcji.

Przedtawiony proces jest wykorzystywany podczas treningu w zaimplementowanych na potrzeby projektu modeli. Za załadowanie do modelu danych treningowych oraz przeprowadzenie eksperymentów odpowiadają funkcje przedstawione w kodzie źródowym 2.7

```
# zaimportowanie niezbędnych bibliotek
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# wybór procesora lub karty graficznej jeżeli jest dostępna
# w celu wykonywania obliczeń
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# preds: tablica wartości zwróconych przez sieć
# labels: tablica z etykietami cyfr do rozpoznania
def get_num_correct(preds, labels):

    #  |
    #  |
    # dim = 0
    #  |
    #  |
    #   ---- dim = 1 --
    # argmax(dim=1): zwraca indeksy
    # maksymalnych wartośći z każdego wiersza
    # funkcja get_num_correct zwraca sumę predykcji
    # wykonanych przez model, które zgadzają się z etykietami
    return preds.argmax(dim=1).eq(labels).sum().item()

# network: model sieci przekazan jakoa rgument funkcji
# epochs: ilość powtórzeń na całym przekazanym zbiorze treningowym
# device: urządzenie, ne którym wykonywane będą obliczenia
def train_net(network, epochs, device):

    # optimizer: jest to algorytm odpowiedzialny za dostosowywanie
    # wag oraz współczynnika uczenia sieci neuronowej
    # Adam: to jeden z dostarczanych z bibliotegi algorytmów wstecznej propagacji
    # network.parameters(): parametry sieci takie jak wagi oraz wartości neuronów
    # lr: współczynnik uczenia sieci
    optimizer = optim.Adam(network.parameters(), lr=0.001)

    # epoch: odpowiada jednorazowemu przejściu przez caly zbiór treningowy
    for epoch in range(epochs):

        total_correct = 0
```

```
        # batch: zbiór obrazów i ich etykiet przekazanych do treningu
        for batch in train_loader:
            images, labels = batch
            images, labels = images.to(device), labels.to(device)

            # preds: wartości zwrócone przez model
            preds = network(images)

            # przypsianie starty między predykcjami zwróconymi przez sieć
            # a aktualnymi wynikami
            # cross_entropy(): funkcja zwracająca miarę niedopasowania wyników
            # im zwrócona wartość jest niższa tym dokładniejsze są wyniki
            loss = F.cross_entropy(preds, labels)

            # zerowanie wartości wektora pochodnych cząstkowych funkcji błędu
            # wektor ten wykorzystywany jest do aktualizowania wag w modelu
            # wksazuje nachylenie funkcji błędu i powala go zminimalizować
            optimizer.zero_grad()

            # propagacja wsteczna
            loss.backward()

            # zaktualizowanie wartości wag modelu
            optimizer.step()

            # kumulowanie poprawnych predykcji
            total_correct += get_num_correct(preds, labels)

        # obliczanie dokładości predykcji sieci
        train_accuracy = round(100. * (total_correct / len(train_loader.dataset)), 4)

        print(f"Results for train network in epoch: {epoch}")
        print(f"Accuracy on training set: {train_accuracy}")
```

Kod źródłowy 2.7: Funkcje odpowiedzialne za trenowanie sieci

Kod źródłowy 2.7 Zaimplementowane zostały: funkcja sprawdzająca poprawność predykcji sieci, przez porównanie wartości zwróconych przez sieć z etykietą rozpoznawanej cyfry, funkcja odpowiedzialna za trening modelu sieci z określoną liczbą iteracji treningowych.

2.5 Szyfrowanie sieci

Szyfrowanie homomorficzne odbywa się za pomocą biblioteki Pyfhel. Schemat prostego zaszyfrowania liczb został przedstawiony na listingu 2.8

```
# zaimportowanie niezbędnych bibliotek
from Pyfhel import Pyfhel, PyPtxt, PyCtxt

# zainicjowanie obiektu klasy
HE = Pyfhel()

# contextGen(): metoda niezbędna do wygenerowania funkcji
# takich jak: zakodowanie do postaci wielomianu, odkodowanie, zaszyfrowanie
# odszyfrowanie, operacje mnożenia i dodawania homomorficznego
# podczas jej zainicjowania generowane są parametry:
# p: na bazie, którego wykonywane będą operacje modulo p
# m: wykładnik wielomianu, który zostanie wykorzystany do
# operacji modulo x^m + 1
HE.contextGen(p=65537, m=4096)

# keyGen(): funkcja generująca parę kluczy: prywatny, publiczny
HE.keyGen()


a = 127.15717263
```

```
b = -2.128965182

# encryptFrac(): szyfruje pojedynczą wartość
ctxt1 = HE.encryptFrac(a)
ctxt2 = HE.encryptFrac(b)

# operacje:
# dodawania
# odejmowania
# mnożenia
# na zaszyfrowanych danych
ctxtSum = ctxt1 + ctxt2
ctxtSub = ctxt1 - ctxt2
ctxtMul = ctxt1 * ctxt2

# decryptFrac(): odszyfrowanie wartości
sum = HE.decryptFrac(ctxtSum)
sub = HE.decryptFrac(ctxtSub)
mul = HE.decryptFrac(ctxtMul)


m1 = HE.encryptFrac(a)

# noiseLevel(): wartość poziomu do którego wzrosnąć może zaszumienie
# nie powodując przy tym utraty danych
noise_room = HE.noiseLevel(m1)

# przedstawienie wyników w konsoli wyjściowej
print(HE)
print(f"Expected sum: {a+b}, decrypted sum: {sum}")
print(f"Expected sub: {a-b}, decrypted sub: {sub}")
print(f"Expected mul: {a*b}, decrypted mul: {mul}")
print(f"Noise room: {noise_room}")
print(HE.noiseLevel(m1+m1))
print(HE.noiseLevel(m1*m1))
print(HE.noiseLevel(m1+m2))
print(HE.noiseLevel(m1*m2))
```

Kod źródłowy 2.8: Zastosowanie biblioteki Pyfhel (przykład)

Kod źródłowy 2.8 Przedstawione zostało szyfrowanie homomorficzne prostych wartości liczbowych oraz operacje mnożenia, dodawania i odejmowania wartości zaszyfrowanych. Porównano wyniki działań na liczb niezaszyfrowanych z liczbami zaszyfrowanymi.

Na listingu 2.9 przedstawiono wartości wyjściowe w konsoli.

```
# parametry obiektu HE
<Pyfhel obj at 0x7f0de4174b50, [pk:Y, sk:Y, rtk:-, rlk:-, contx(p=65537, m=4096, base=2,
    sec=128, dig=64i.32f, batch=False)]>

# porównanie wynik sumy uzyskanego z odszyfrowanych danych
Expected sum: 125.028207448, decrypted sum: 125.02820744784549

# porównanie wyniku odejmowania uzyskanego z odszyfrowanych danych
Expected sub: 129.286137812, decrypted sub: 129.28613781183958

# porównanie wyniku mnożenia uzyskanego z odszyfrowanych danych
Expected mul: -270.7131931708334, decrypted mul: -270.7131931686308

# poziom możliwego błędu zaszumienia dla operacji zaszyfrowania m1
82

# poziom dostępnego błędu zaszumienia dla operacji m1 + m1
81

# poziom dostępnego błędu zaszumienia dla operacji m1 * m1
# w przypadku mnożenia większych wartości
# margines dostępnego błędu spada
54
```

```
# poziom możliwego błędu zaszumienia dla operacji m1 + m2
82

# poziom możliwego błędu zaszumienia dla operacji m1 * m2
82
```

Kod źródłowy 2.9: Wyniki kodu 2.8

Zaszyfrowane modele sieci, wykonują obliczenia na macierzach zawierających dane, które również poddane są operacji szyfrowania homomorficznego, dodatkowo wartości wag w modelach także są reprezentowane za pomocą macierzy. Funckje odpowiedzialne za proces zakodowania macierzy danych do postaci macierzy wielomianów, oraz zaszyfrowania homomorficznego macierzy zostały przedstawione w kodzie źródłowym 2.10

```
# funkcja kodująca macierz danych do postaci macierzy wielomianów
def encode_matrix(HE, matrix):
    try:
        # zwraca zakodowaną tablicę elementów
        # map(funkcja, argument): mapuje funkcję kodującą
        # na każdy element podanego argumentu
        # list(): tworzy listę obiektów
        # np.array(): tworzy tablicę biblioteki numpy
        return np.array(list(map(HE.encodeFrac, matrix)))

    # w przypadku niewłaściwego typu podanego w pierwszej części funckji
    # odbywa się przekazywanie jako argumentu wierszy macierzy (m in matrix)
    except TypeError:
        # zwrócona zostaje tablica, której elementami są tablice czyli macierz
        # następuje wywołanie rekurencyjne funkcji (wywołuje ona samą siebie)
        return np.array([encode_matrix(HE, m) for m in matrix])

# funkcja dekodująca macierz wielomianó do postaci macierzy danych
def decode_matrix(HE, matrix):
    try:
        return np.array(list(map(HE.decodeFrac, matrix)))
    except TypeError:
        return np.array([decode_matrix(HE, m) for m in matrix])

# funkcja szyfrująca macierz wielomianów
def encrypt_matrix(HE, matrix):
    try:
        # zwraca zaszyfrowaną tablicę elementów
        # map(funkcja, argument): mapuje funkcję szyfrującą
        # na każdy element podanego argumentu
        # list(): tworzy listę obiektów
        # np.array(): tworzy tablicę biblioteki numpy
        return np.array(list(map(HE.encryptFrac, matrix)))

    # w przypadku niewłaściwego typu podanego w pierwszej części funckji
    # odbywa się przekazywanie jako argumentu wierszy macierzy (m in matrix)
    except TypeError:
        # zwrócona zostaje tablica, której elementami są tablice, czyli macierz
        # następuje wywołanie rekurencyjne funkcji (wywołuje ona samą siebie)
        return np.array([encrypt_matrix(HE, m) for m in matrix])

# funkcja deszyfrująca zaszyfrowaną macierz
def decrypt_matrix(HE, matrix):
    try:
        return np.array(list(map(HE.decryptFrac, matrix)))
    except TypeError:
        return np.array([decrypt_matrix(HE, m) for m in matrix])
```

Kod źródłowy 2.10: Funkcje kodujące macierz danych do postaci macierzy wielomianów, oraz szyfrujące macierz danych zakodowanych

Kod źródłowy 2.10 Przedstawione zostały funkcje odpowiedzialne za: kodowanie macierzy danych do postaci wielomianu, odkodowanie macierzy danych, zaszyfrowanie macierzy danych, odszyfrowanie macierzy danych.

2.5.1 Szyfrowanie po warstwach sieci

Zaszyfrowanie modeli sieci odbywa się poprzez zmapowanie funkcji szyfrującej na każdą z warstw modelu. W tym celu zbudowane zostały klasy odpowiadające każdej warstwie występującej w modelach, posiadające funkcje, odpowiedzialne za wykorzystanie biblioteki szyfrowania homomorficznego. Klasy zostały przedstawione w kodach źródłowych 2.11, 2.12, 2.13, 2.14 i 2.15.

```
class ConvolutionalLayer:
    # konstruktor klasy, przyjmuje jako parametry:
    # HE: obiekt biblioteki Pyfhel
    # weights: wagi modelu
    # stride: krok operacji konwolucji
    # padding: obaszr obramowania jaki posiadają dane wejściowe
    # bias: bias wykorzystywany jako jeden z zsumowanych argumentów funkcji aktywacyjnej
    def __init__(self, HE, weights, stride=(1, 1), padding=(0, 0), bias=None):
        self.HE = HE
        # przypisywane wartości wag są w postaci wielomianów
        self.weights = encode_matrix(HE, weights)
        self.stride = stride
        self.padding = padding
        self.bias = bias
        if bias is not None:
            self.bias = encode_matrix(HE, bias)

    # __call__: pozwala na wykonanie zaimplementowanych działań
    # podczas wywołania obiektu klasy
    def __call__(self, t):
        # aplikuje obszar obramowania do obrazu
        t = apply_padding(t, self.padding)

        # result: wartość zwracana przez __call__()
        # przypisana zostaje macierz uzyskana po operacji konwolucji
        # ponieważ sum() obywa się po osi pionowej, wartości z poszczególnych
        # operacji konwolucji zostają złączone w wiersz macierzy
        # image_layer: warstwa danych z obrazu
        # filter_layer: warstwa filtra mapującego
        result = np.array([[np.sum([convolute2d(image_layer, filter_layer, self.stride)
                                    for image_layer, filter_layer in zip(image, _filter)
    ], axis=0)
                            # _filter: wagi filtra mapującego
                            for _filter in self.weights]
                           for image in t])

        # jeżeli bias posiada przypisaną wartość, zostaje ona dodana do zmapowanych wynik
    ów
        if self.bias is not None:
            return np.array([[layer + bias for layer, bias in zip(image, self.bias)] for
    image in result])
        # w przeciwnym wypadku zwracana zostaje wartość: result
        else:
            return result

# funkcja odpowiedzialna za operację konwolucji
def convolute2d(image, filter_matrix, stride):
    # rozmiar poziomy macierzy danych obrazu
    x_d = len(image[0])
    # rozmiar pionowy macierzy danych obrazu
    y_d = len(image)

    # rozmiar poziomy macierzy filtra
    x_f = len(filter_matrix[0])
```

```
    # rozmiar pionowy macierzy filtra
    y_f = len(filter_matrix)

    # wartość kroku pionowego operacji
    y_stride = stride[0]
    # wartość kroku poziomego operacji
    x_stride = stride[1]

    # zakres poziomych kroków operacji
    x_o = ((x_d - x_f) // x_stride) + 1
    # zakres pionowych kroków operacji
    y_o = ((y_d - y_f) // y_stride) + 1

    # funkcja aplikująca na obraz filtr mapujący
    def get_submatrix(matrix, x, y):
        # od indeksu y w bieżącym kroku operacji
        index_row = y * y_stride
        # od indeksu x w bieżącym kroku operacji
        index_column = x * x_stride
        # zwracana jest macierz obrazu pokryta wymiarami filtra mapującego
        return matrix[index_row: index_row + y_f, index_column: index_column + x_f]

    # zwracana jest wartość wynikająca z operacji konwolucji
    return np.array(
        [[np.sum(get_submatrix(image, x, y) * filter_matrix) for x in range(0, x_o)] for
    y in range(0, y_o)])

# funckja aplikująca obramowanie, obramowanie wykorzystywane
# jest aby zredukować utratę danych przy małych obrazach
# ponieważ operacja konwolucji skutkuje coraz mniejszymi
# mapami przetwarzanego obrazu,
# wartości paddingu są równe 0 tak aby nie wpływały
# na wynik operacji konwolucji
# w przypadku danych ze zbioru
# MNIST obramowanie nie występuje
# jednak ta funkcja jest częścią klasy z biblioteki PyTorch
# dlatego dla spójności obiektów została zaimpelementowana
def apply_padding(t, padding):
    # wartość pionowego zakresu obramowania
    y_p = padding[0]
    # wartość poziomego zakresu obramowania
    x_p = padding[1]

    # wartość zero w obramowaniu
    zero = t[0][0][y_p+1][x_p+1] - t[0][0][y_p+1][x_p+1]
    # zwraca obraz z obramowaniem
    return [[np.pad(mat, ((y_p, y_p), (x_p, x_p)), 'constant', constant_values=zero) for
    mat in layer] for layer in t]
```

Kod źródłowy 2.11: Klasa reprezentująca zaszyfrowaną warstwę konwolucyjną

Kod źródłowy 2.11 Zaimplementowana została warstwa konwolucyjna sieci, zgodnie z architekturą w bibliotece PyTorch. Klasa która została zaimplementowana umożliwia zaszyfrowanie homomorficzne wag połączeń między neuronami. A działania wykonywane w niej są działaniami na danych zaszyfrowanych.

```
class LinearLayer:
    # konstruktor klasy, przyjmuje jako parametry:
    # HE: obiekt biblioteki Pyfhel
    # weights: wagi modelu
    # bias: bias wykorzystywany jako jeden z zsumowanych argumentów funkcji aktywacyjnej
    def __init__(self, HE, weights, bias=None):
        self.HE = HE
        self.weights = encode_matrix(HE, weights)
        self.bias = bias
        if bias is not None:
            self.bias = encode_matrix(HE, bias)

    # __call__: podczas wywołania obiektu klasy zwraca wartość result
```

```
    def __call__(self, t):
        # result: tablica zawierająca argumenty dla kolejnej warstwy
        result = np.array([[np.sum(image * row) for row in self.weights] for image in t])
        if self.bias is not None:
            # jeżeli bias istnieje jest dodawany do elementów tablicy argumentów
            result = np.array([row + self.bias for row in result])
        return result
```

Kod źródłowy 2.12: Klasa reprezentująca zaszyfrowaną warstwę Liniową

Kod źródłowy 2.12 Zaimplementowana została warstwa liniowa sieci, zgodnie z architekturą w bibliotece PyTorch. Klasa która została zaimplementowana umożliwia zaszyfrowanie homomorficzne wag połączeń między neuronami. A działania wykonywane w niej są działaniami na danych zaszyfrowanych.

```
class SquareLayer:
    # konstruktor klasy, przyjmuje jako parametry:
    # HE: obiekt biblioteki Pyfhel
    def __init__(self, HE):
        self.HE = HE

    # __call__: podczas wywołania obiektu klasy zwraca listę argumentów podniesionych do
    kwadratu
    def __call__(self, image):
        return square(self.HE, image)

# funckja wykonująca operację podniesienia zaszyfrowanych argumentów do potęgi drugiej
def square(HE, image):
    try:
        # zwrócenie tablicy zaszyfrowanych argumentów podniesionych do potęgi drugiej
        return np.array(list(map(lambda x: HE.power(x, 2), image)))
    except TypeError:
        # w przypadku niezgodności argumentów z typem danych przyjmowanych przez funkcję
        # następuje wywołanie rekurencyjne funkcji na elementach argumentu
        return np.array([square(HE, m) for m in image])
```

Kod źródłowy 2.13: Klasa reprezentująca zaszyfrowaną funkcję aktywacyjną x^2

Kod źródłowy 2.13 Zaimplementowana została warstwa sieci, pozwalająca na skorzystanie z funkcji aktywacyjnej *Square*(), działania wykonywane w niej są działaniami na danych zaszyfrowanych.

```
class FlattenLayer:
    # __call__: podczas wywołania obiektu klasy zwraca jednowymiarową tablicę danych
    def __call__(self, image):
        dimension = image.shape
        return image.reshape(dimension[0], dimension[1]*dimension[2]*dimension[3])
```

Kod źródłowy 2.14: Klasa reprezentująca warstwę spłaszczającą dane

Kod źródłowy 2.14 Zaimplementowana została klasa reprezentująca warstwę spłaszczająca dane wejściowe do postaci tablicy jednowymiarowej.

```
class AveragePoolLayer:
    # konstruktor klasy, przyjmuje jako parametry:
    # HE: obiekt biblioteki Pyfhel
    # kernel_size: rozmiar filtra próbkującego
    # stride: zakres kroku operacji
    # padding: wymiar obramowania obrazu
    def __init__(self, HE, kernel_size, stride=(1, 1), padding=(0, 0)):
        self.HE = HE
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

    # __call__: podczas wywołania obiektu klasy:
    # aplikuje obramowanie obrazu (jesli istnieje)
    # zwraca  macierz powstałą na skutek operacji próbkowania
    def __call__(self, t):
        t = apply_padding(t, self.padding)
```

```
        return np.array([[_avg(self.HE, layer, self.kernel_size, self.stride) for layer
    in image] for image in t])

# funkcja opdowiedzialana za przeprowadzenie
# operacji próbkowania średnich
def _avg(HE, image, kernel_size, stride):
    # wartość kroku poziomego operacji
    x_s = stride[1]
    # wartość kroku pionowego operacji
    y_s = stride[0]

    # poziomy rozmiar filtra
    x_k = kernel_size[1]
    # pionowy rozmiar filtra
    y_k = kernel_size[0]

    # poziomy rozmiar obrazu
    x_d = len(image[0])
    # poziomy rozmiar filtra
    y_d = len(image)

    # poziomy zakres kroków operacji
    x_o = ((x_d - x_k) // x_s) + 1
    # pionowy zakres kroków operacji
    y_o = ((y_d - y_k) // y_s) + 1

    # ułamek służący do wyliczenia średniej
    denominator = HE.encodeFrac(1 / (x_k * y_k))

    # zwraca macierz pokrytą przez filtr próbkujący
    def get_submatrix(matrix, x, y):
        index_row = y * y_s
        index_column = x * x_s
        return matrix[index_row: index_row + y_k, index_column: index_column + x_k]
    # zwraca średnią wartość z filtra
    return [[np.sum(get_submatrix(image, x, y)) * denominator for x in range(0, x_o)] for
     y in range(0, y_o)]
```

Kod źródłowy 2.15: Klasa reprezentująca zaszyfrowaną warstwę próbkującą

Kod źródłowy 2.15 Zaimplementowana została klasa reprezentująca warstwę próbkującą sieci, zgodnie z architekturą w bibliotece PyTorch. Klasa która została zaimplementowana umożliwia zaszyfrowanie homomorficzne wag połączeń między neuronami. A działania wykonywane w niej są działaniami na danych zaszyfrowanych.

Klasy reprezentujące zaszyfrowane warstwy sieci, zostają wykorzystane w funckji, która jako argument przyjmuje wytrenowany obiekt modelu PyTorch. Struktura tej funckji przedstawiona została w kodzie źródłowym 2.16

```
def build_from_pytorch(HE, net):
    # w funkcji zostały zdefiniowane funkcje pomocnicze
    # odpowiedzialne za budowanie zaszyfrowanych warstw modelu
    # jako argumenty przekazywane są:
    # HE: obiekt biblioteki Pyfhel
    # net: model sieci

    # warstwa konwolucyjna
    def conv_layer(layer):
        if layer.bias is None:
            bias = None
        else:
            bias = layer.bias.detach().numpy()

        return ConvolutionalLayer(HE, weights=layer.weight.detach().numpy(),
                                  stride=layer.stride,
                                  padding=layer.padding,
                                  bias=bias)
```

```
    # warstwa liniowa
    def lin_layer(layer):
        if layer.bias is None:
            bias = None
        else:
            bias = layer.bias.detach().numpy()
        return LinearLayer(HE, layer.weight.detach().numpy(),
                           bias)

    # warstwa próbkująca
    def avg_pool_layer(layer):
        # This proxy is required because in PyTorch an AvgPool2d can have kernel_size,
    stride and padding either of
        # type (int, int) or int, unlike in Conv2d
        kernel_size = (layer.kernel_size, layer.kernel_size) if isinstance(layer.
    kernel_size, int) else layer.kernel_size
        stride = (layer.stride, layer.stride) if isinstance(layer.stride, int) else layer
    .stride
        padding = (layer.padding, layer.padding) if isinstance(layer.padding, int) else
    layer.padding

        return AveragePoolLayer(HE, kernel_size, stride, padding)

    # warstwa spłaszczająca
    def flatten_layer(layer):
        return FlattenLayer()

    # warstwa funkcji aktywacyjnej
    def square_layer(layer):
        return SquareLayer(HE)

    # mapowanie odpowiednich funckji budujących
    # w zależności od budowy przekazanego modelu
    options = {"Conv": conv_layer,
               "Line": lin_layer,
               "Flat": flatten_layer,
               "AvgP": avg_pool_layer,
               "Squa": square_layer
               }

    # zwracane są warstwy zaszyfrowane
    encoded_layers = [options[str(layer)[0:4]](layer) for layer in net]
    return encoded_layers
```

Kod źródłowy 2.16: Funkcja budująca zaszyfrowany model

Kod źródłowy 2.16 Zaimplementowana została funkcja, która buduje model sieci zaszyfrowany bazując na modelu podanym jako argument wejściowy, zawiera funkcje pomocnicze odpowiedzialne za zaszyfrowanie odpowiednich warstw.

2.6 Testowanie sieci

Do testowania sieci niezaszyfrowanych zaimplementowana została funkcja przedstawiona w kodzie źródłowym 2.17

```
def test_net(network, device):
    # jako parametry przyjmowane są:
    # network: model sieci
    # device: definiuje czy obliczenia wykonywane są na CPU czy na GPU

    # zmienia tryb sieci z treningowego na testowy
    network.eval()

    # zmienna do zliczania poprawnych predykcji
```

```
    total_correct = 0

    # w celu zaoszczędzenia pamięci
    # wyłączono obliczania gradientu
    with torch.no_grad():
        # przekazanie części zbioru testowego
        for batch in test_loader:
            images, labels = batch
            images, labels = images.to(device), labels.to(device)

            # predykcje uzyskane przez sieć
            preds = network(images)

            # zliczanie prawidłowych predykcji
            total_correct += get_num_correct(preds, labels)

        # procentowe przedstawienie precyzji sieci
        accuracy = round(100. * (total_correct / len(test_loader.dataset)), 4)

    return total_correct / len(test_loader.dataset)
```

Kod źródłowy 2.17: Funkcja testująca sieć niezaszyfrowaną

Kod źródłowy 2.17 Zaimplementowana została funkcja odpowiedzialna za przeprowadzenie testów na modelu sieci niezaszyfrowanym, wykorzystuje niezaszyfrowane dane ze zbioru testowego MNIST.

Do testowania sieci zaszyfrowanych zaimplementowana została funkcja przedstawiona w kodzie źródłowym 2.18

```
# funkcja testująca modele sieci
def test_parameters(n, p, model):
    # jako paramtery przyjmuje
    # p: wartość do działania modulo na współczynnikach wielomianów
    # n: wartość najwyższego stopia wielomianu
    # model: model sieci

    # zainicjowanie obiektu szyfrującego homomorficznie
    HE = Pyfhel()
    HE.contextGen(p=p, m=n)
    HE.keyGen()

    # zdefiniowanie rozmiaru wielomianu przy którym zachodzi realineryzacja
    relinKeySize=3
    HE.relinKeyGen(bitCount=2, size=relinKeySize)

    # ładowanie obrazów oraz ich etykiet
    images, labels = next(iter(test_loader))

    sample_image = images[0]
    sample_label = labels[0]

    # zdefiniowanie procesora do wykonywania obliczeń
    model.to("cpu")
    model_encoded = build_from_pytorch(HE, model)

    with torch.no_grad():
        # wartość zwrócona przez model niezaszyfrowany
        expected_output = model(sample_image.unsqueeze(0))

    # zaszyfrowanie obrazu
    encrypted_image = encrypt_matrix(HE, sample_image.unsqueeze(0).numpy())

    # przekazywanie zaszyfrowanego obrazu do kolejnych warstw sieci zaszyfrowanej
    for layer in model_encoded:
        encrypted_image = layer(encrypted_image)
        print(f"Passed layer {layer}...")

    # wynik działania sieci zaszyfrowanej
```

```
    result = decrypt_matrix(HE, encrypted_image)

    # różnica w wynikach między modelami
    difference = expected_output.numpy() - result
```

Kod źródłowy 2.18: Funkcja testująca sieć zaszyfrowaną

Kod źródłowy 2.18 Zaimplementowana została funkcja odpowiedzialna za przeprowadzenie testów na modelu sieci zaszyfrowanym, wykorzystuje zaszyfrowane dane ze zbioru testowego MNIST. W funkcji został zawarty proces rozpoznawania jednego zaszyfrowanego obrazu ze zbioru MNIST. Do wykonania eksperymentów na większej ilości danych, funkcja zostanie zmodyfikowana.

Rozdział 3

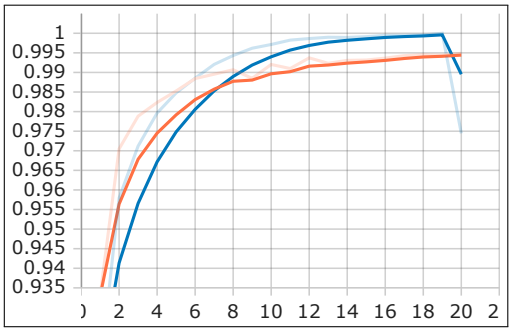
Testowanie i wyniki

W tym rozdziale, przedstawiono wyniki eksperymentów przeprowadzonych w platformie Google Colab i odpowiadające im wnioski.

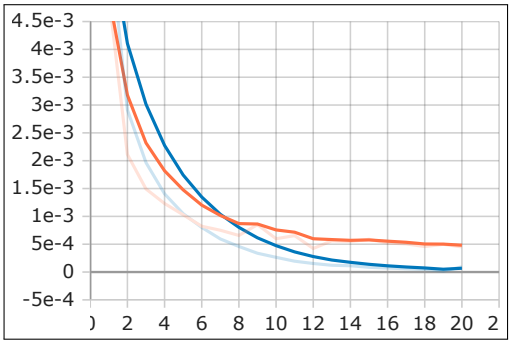
3.1 Test sieci neuronowych

3.1.1 Model liniowy niezaszyfrowany

Przetestowano model liniowy. W procesie treningowym zmierzono różnice między siecią wykorzystującą funkcję aktywacyjną$tanh()$ dwukrotnie, oraz siecią korzystającą z funkcji aktywacyjnej $Square()$ dwukrotnie. Na grafice 3.1a przedstawiono wykres precyzji, a na grafice 3.1b wykres funkcji straty, które uzyskano podczas treningu tych sieci.



(a) Wykres precyzji



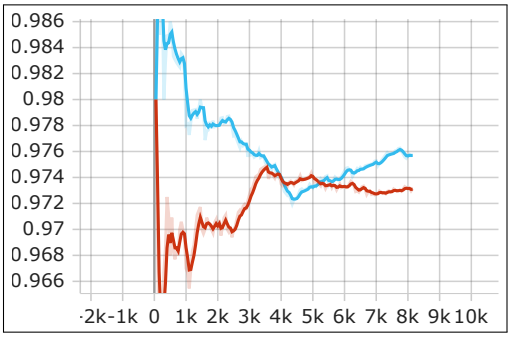
(b) Wykres funkcji straty

Rysunek 3.1: Wykres precyzji i funkcji starty podczas 20 iteracji treningu sieci liniowych niezaszyfrowanych, kolorem niebieskim oznaczono model z funkcją $tanh()$, kolorem pomarańczowym model z funkcją $Square()$.

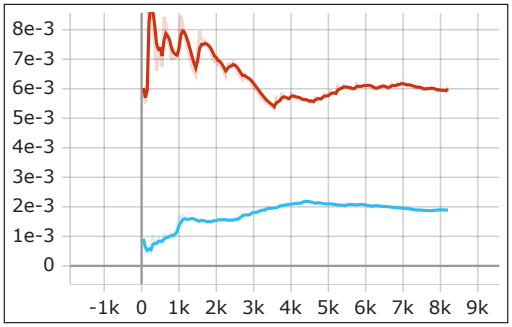
Czas treningu	Precyzja	Wartość funkcji straty	Model sieci	Funkcja aktywacyjna
3min 24s	99.85%	$9.87*(10)^{-5}$	Liniowy	dwukronie $tanh()$
3min 41s	99.49%	$4.5*(10)^{-4}$	Liniowy	dwukrotnie $Square()$

Tablica 3.1: Tabela przedstawia różnice w wynikach po treningu, między modelem liniowym nieszyfrowanym korzystającym z funkcji aktywacyjnej $tanh()$, a modelem liniowym nieszyfrowanym korzystającym z funkcji aktywacjnej $Square()$.

Wytrenowane modele zostały użyte na zbiorach testowych, wyniki uzyskane podczas tego procesu przedstawione zostały na grafikach 3.2a oraz 3.2b.



(a) Wykres precyzji



(b) Wykres funkcji straty

Rysunek 3.2: Wykres precyzji i funkcji starty podczas testu sieci liniowych niezaszyfrowanych na zbiorze testowym MNIST, kolorem niebieskim oznaczono model z funkcją *tanh*(), kolorem pomarańczowym model z funkcją *Square*().

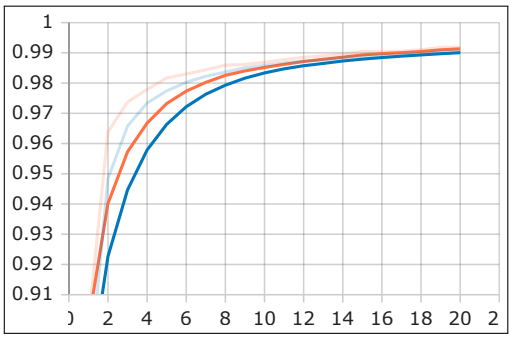
Czas testu	Precyzja	Strata	Model	Funkcja aktywacyjna
1.28s	97.6%	$1.99*(10)^{-3}$	Liniowy	tanh()
1.21s	97.3%	$6*(10)^{-3}$	Liniowy	Square()

Tablica 3.2: Tabela przedstawia różnice w wynikach testu, między modelem liniowym niezaszyfrowanym korzystającym z funkcji aktywacyjnej *tanh*(), a modelem liniowym niezaszyfrowanym korzystającym z funkcji aktywacjnej *Square*().

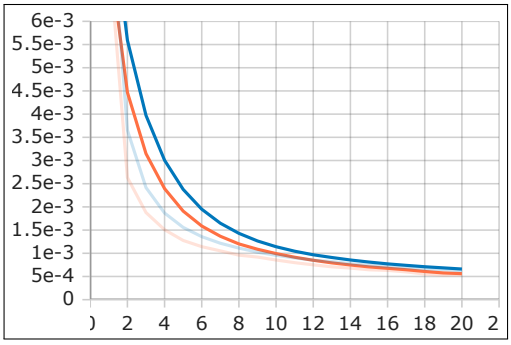
Czasy treningu i czasy testu dla obydwu modeli są zbliżone. Model sieci korzystający z funckji aktywującej *Square*(), ma wyższe wartości funkcji straty niż model korzystający z funkcji *tanh*(), jednak precyzja obydwu modeli na zbiorze treningowym i zbiorze testowym jest bardzo zbliżona, to potwierdza, że model z funkcją *Square*() może zostać użyty zamiast modelu korzystającego z funkcji *tanh*() i uzyskać podobną precyzję.

3.1.2 Model LeNet1 niezaszyfrowany

Przetestowano model LeNet1. W procesie treningowym zmierzono różnice między siecią wykorzystującą funkcję aktywacyjną*tanh*() dwukrotnie, oraz siecią korzystającą z funkcji aktywacyjnej *Square*() dwukrotnie. Na grafice 3.3a przedstawiono wykres precyzji, a na grafice 3.3b wykres funkcji straty, które uzyskano podczas treningu tych sieci.



(a) Wykres precyzji



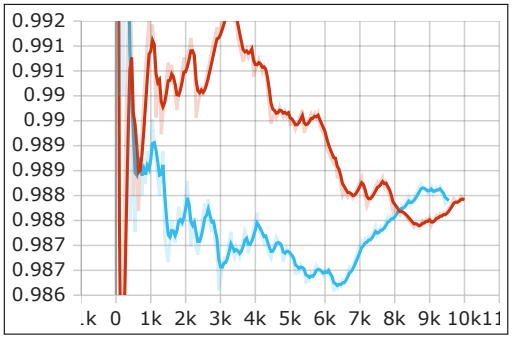
(b) Wykres funkcji straty

Rysunek 3.3: Wykres precyzji i funkcji starty podczas 20 iteracji treningu sieci LeNet1 niezaszyfrowanych, kolorem niebieskim oznaczono model z funkcją *tanh*() wykorzystaną dwukrotnie, kolorem pomarańczowym model z funkcją *Square*() wykorzystaną dwukrotnie.

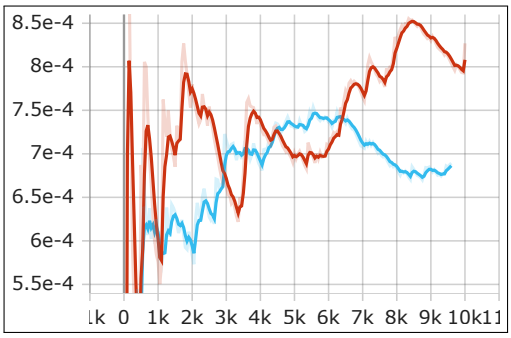
Czas treningu	Precyzja	Strata	Model	Funkcja aktywacyjna
5min 39s	99.05%	$6.2*(10)^{-4}$	LeNet1	tanh()
5min 41s	99.18%	$5.3*(10)^{-4}$	LeNet1	Square()

Tablica 3.3: Tabela przedstawia różnice w wynikach po treningu, między modelem LeNet1 niezaszyfrowanym korzystającym dwukrotnie z funkcji aktywacyjnej *tanh*(), a modelem liniowym niezaszyfrowanym korzystającym dwukrotnie z funkcji aktywacyjnej *Square*().

Wytrenowane modele zostały użyte na zbiorach testowych, wyniki uzyskane podczas tego procesu przedstawione zostały na grafikach 3.4a oraz 3.4b.



(a) Wykres precyzji



(b) Wykres funkcji straty

Rysunek 3.4: Wykres precyzji i funkcji starty podczas testu sieci LeNet1 niezaszyfrowanych na zbiorze testowym MNIST, kolorem niebieskim oznaczono model z funkcją *tanh*() wykorzystaną dwukrotnie, kolorem pomarańczowym model z funkcją *Square*() wykorzystaną dwukrotnie.

Czas testu	Precyzja	Średnia strata	Model	Funkcja aktywacyjna
1.69s	98.8%	$6.9*(10)^{-4}$	LeNet1	tanh()
1.6s	98.8%	$8.1*(10)^{-4}$	LeNet1	Square()

Tablica 3.4: Tabela przedstawia różnice w wynikach testu, między modelem LeNet1 niezaszyfrowanym korzystającym dwukrotnie z funkcji aktywacyjnej *tanh*(), a modelem liniowym niezaszyfrowanym korzystającym dwukrotnie z funkcji aktywacjnej *Square*().

Czasy treningu i czasy testu dla obydwu modeli są zbliżone. Model sieci korzystający z funckji aktywującej *Square*(), ma wyższą średnią wartość funkcji straty podczas testu niż model korzystający z funkcji *tanh*(), jednak precyzja obydwu modeli na zbiorze treningowym i zbiorze testowym jest bardzo zbliżona, z przewagą modelu z funkcją aktywcyjną *Square*() to potwierdza, że model z funkcją *Square*() może zostać użyty zamiast modelu korzystającego z funkcji *tanh*() i uzyskać podobną precyzję.

3.1.3 Zaszumienie przy szyfrowaniu modelu LeNet1 z dwiema funkcjami aktywacyjnymi *Square*().

Zaszyfrowano model liniowy, z funkcją aktywacyjną *Square*() i sprawdzono wynik jego predykcji na zaszyfrowanym obrazie, wynik uzyskany w konsoli przedstawiono na listingu 3.1.

```
# predykcje modelu zaszyfrowanego
Decrypted preds: tensor([[   9.3001, -123.9066,  -61.9963,  -51.4525,  -60.2537,
    -7.4224,
           46.7155,  -45.3799,  -21.0661,  -64.5128]])

# prawidłowa etykieta
Actual label: tensor([6])

# predykcje modelu niezaszyfrowanego
Actual preds: [tensor([[   9.3085, -123.9135,  -62.0036,  -51.4597,  -60.2539,    -7.4216,
           46.7238,  -45.3823,  -21.0672,  -64.5181]],
       grad_fn=<AddmmBackward0>)]

Total correct: 1
```

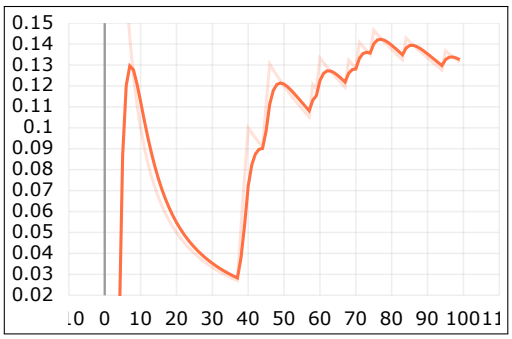
Kod źródłowy 3.1: Wynik predykcji zwróconych przez model liniowy z funkcją aktywacyjną *Square*() zaszyfrowany, oraz aktualny wynik zwrócony przez ten sam model niezaszyfrowany

Zaszyfrowano model LeNet1, z dwiema funkcjami aktywacyjnymi *Square*() i sprawdzono wynik jego predykcji na zaszyfrowanym obrazie, takim jakiego użyto dla zaszyfrowanego modelu liniowego, wynik uzyskany w konsoli przedstawiono na listingu 3.2.

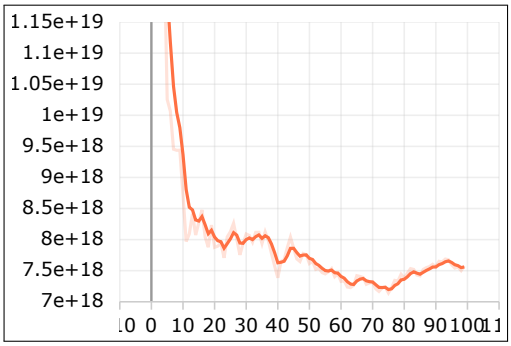
```
# predykcje modelu zaszyfrowanego
Decryted preds: tensor([[ 6.8358e+18, -3.5048e+18,  3.5610e+18,  6.9482e+18, -7.6791e+18,
          6.8400e+18, -7.5921e+18, -8.7907e+18,  7.6502e+18,  3.2230e+18]])

# prawidłowa etykieta
Actual label: tensor([6])

# predykcje modelu niezaszyfrowanego
Actual preds: tensor([[ -4.6989, -19.2495, -15.1032, -11.8074, -17.3342,  -7.1603,
    15.4903,
         -27.9364,  -3.5927, -19.0256]], grad_fn=<AddmmBackward0>)
Total correct: 0
```

Kod źródłowy 3.2: Wynik predykcji zwróconych przez model LeNet1 zaszyfrowany, oraz aktualny wynik zwrócony przez model niezaszyfrowany

Zaszyfrowano model LeNet1, z dwiema funkcjami aktywacyjnymi *Square*() i sprawdzono wynik jego precyzji na 100 zaszyfrowanych obrazach ze zbioru MNIST, wykres precyzji przedstawiono na grafice 3.5a, a wykres funkcji straty na grafice 3.5b. Średnią precyzję, czas, średnią wartość funkcji straty z tego testu przedstawiono w tabeli 3.5



(a) Wykres precyzji



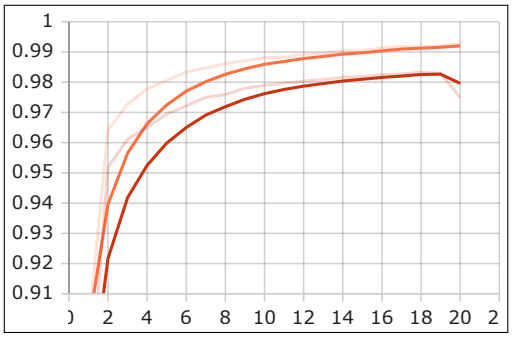
(b) Wykres funkcji straty

Rysunek 3.5: Test na 100 zaszyfrowanych obrazach ze zbioru MNIST, zaszyfrowanego modelu sieci LeNet1 korzystającego z dwóch funkcji aktywacyjnych *Square*().

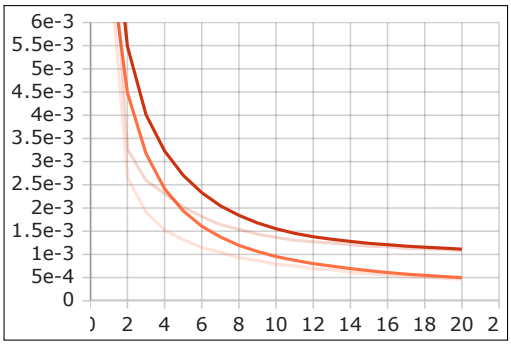
Czas testu	Precyzja	Strata	Model	Funkcja aktywacyjna
4h 54min 18s	13.25%	$7.6*(10)^{18}$	LeNet1	Square()

Tablica 3.5: Tabela przedstawia wyniki uzyskane podczas testu na 100 zaszyfrowanych obrazach ze zbioru MNIST, zaszyfrowanego modelu LeNet1 korzystającego z dwóch funkcji aktywacyjnych *Square*().

Predykcja dokonana przez zaszyfrowany model liniowy okazała się poprawna, jednak predykcja zaszyfrowanego modelu LeNet1 nie zgadza się z predykcją sieci niezaszyfrowanej. Jest to spowodowane zbyt dużym zaszumieniem. Sieć LeNet1, która zostaje poddana operacji szyfrowania, wykorzystuje funckję aktywacyjną *Square*() dwukrotnie. Ponieważ funkcja ta opiera się na mnożeniu, to powoduje duży wynik zaszumienia, a użycie jej dwukrotnie powoduje nieprawidłowy wynik końcowy. Wykorzystanie funkcji aktywacyjnej jednokrotnie niweluje ten problem. Porównanie wyników precyzji i funkcji straty podczas treningu między modelem LeNet1 wykorzystującym funkcję aktywacyjną *Square*() jednokrotnie oraz modelem LeNet1 wykorzystującym tę funkcję dwukrotnie zostało przedstawione na grafikach 3.6a oraz 3.6b.



(a) Wykres precyzji



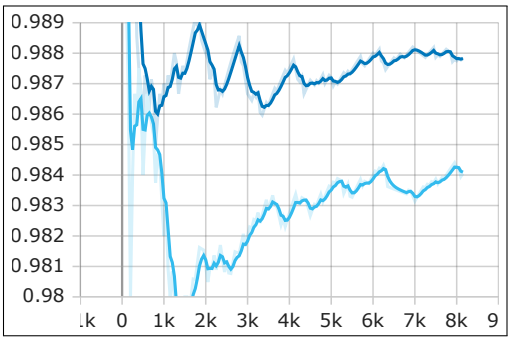
(b) Wykres funkcji straty

Rysunek 3.6: Wykres precyzji i funkcji starty podczas 20 iteracji treningu sieci LeNet1 niezaszyfrowanych, kolorem czerwonym oznaczono model z jedną funkcją aktywacyjną *Square*(), kolorem pomarańczowym model z dwiema funkcjami aktywacyjnymi funkcją *Square*().

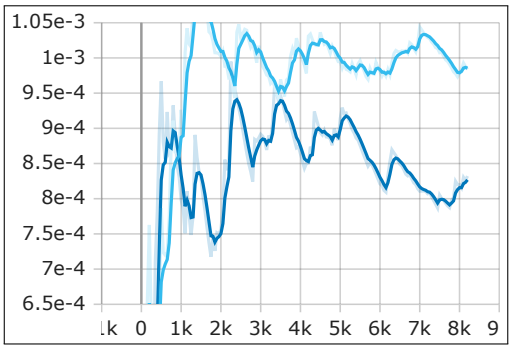
Czas	Precyzja	Strata	Model	Funkcja aktywacyjna
5min 31s	98.32%	$1*(10)^{-3}$	LeNet1	*Square*() użyta raz
5min 37s	99.27%	$4.6*(10)^{-4}$	LeNet1	*Square*() użyta 2 razy

Tablica 3.6: Tabela przedstawia różnice w wynikach uzyskanych po treningu między modelem LeNet1 z z jedną funkcją atywacyjną *Square*() i modelem LeNet1 z dwiema funkcjami aktywacyjnymi *Square*().

Wytrenowane modele zostały użyte do przeprowadzenia predykcji na zbiorach testowych, a wyniki uzyskane podczas tego procesu przedstawione zostały na grafikach 3.7a oraz 3.7b.



(a) Wykres precyzji



(b) Wykres funkcji straty

Rysunek 3.7: Wykres precyzji i funkcji starty podczas testu sieci LeNet1 niezaszyfrowanych, kolorem jasnoniebieskim oznaczono model z jedną funkcją aktywacyjną *Square*(), kolorem ciemnoniebieskim model z dwiema funkcjami aktywacyjnymi funkcją *Square*().

Czas	Precyzja	Strata	Model	Funkcja aktywacyjna
1.57s	98.19%	$1.1*(10)^{-3}$	LeNet1	*Square*() użyta raz
1.67s	98.62%	$9.3*(10)^{-4}$	LeNet1	*Square*() użyta 2 razy

Tablica 3.7: Tabela przedstawia różnice w wynikach uzyskanych podczas testu na zbiorze testowym MNIST między modelem LeNet1 z jedną funkcją atywacyjną *Square*() i modelem z dwiema funkcjami aktywacyjnymi *Square*().

Czas działania obydwu sieci jest zbliżony. Wartość funkcji strat jest większa w przypadku modelu LeNet1 wykorzystującego jedną funkcję aktywacyjną, jednka precyzja uzyskiwana przez obydwa modele jest bardzo zbliżona. Oznacza to, że model z jedną funkcją aktywacyjną *Square*() można wykorzystać zamiast modelu LeNet1 korzystającego z tej funkcji dwukrotnie oraz uzyskać przy tym bardzo zbliżoną precyzję. Wynik predykcji uzyskane przez zaszyfrowany model z jedną funkcją aktywacyjną został przedstawiony na listingu 3.3

```
# predykcje modelu zaszyfrowanego
Decryted preds: tensor([[ -4.5184, -14.5355,  -7.7539,  -5.6933,  16.9160,  -2.0540,
    -3.4083,
          0.7825,   3.3187,   8.8705]])

# prawidłowa etykieta
Actual label: tensor([4])
```

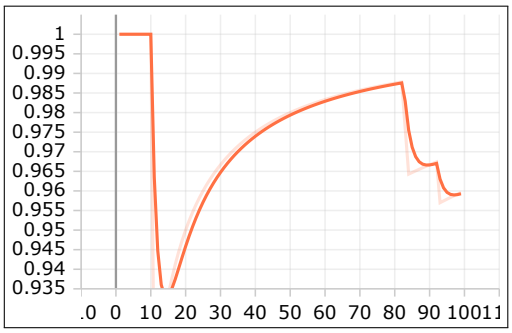
```
# predykcje modelu niezaszyfrowanego
Actual preds: tensor([[ -4.5365, -14.5714,  -7.7614,  -5.7148,  16.9779,  -2.0561,
    -3.4146,
10           0.7813,   3.3146,   8.8931]], grad_fn=<AddmmBackward0>)

Total correct: 1
```

Kod źródłowy 3.3: Wynik predykcji zwróconych przez model LeNet1 z jedną funkcją aktywacyjną *Square*() zaszyfrowany, oraz aktualny wynik zwrócony przez model niezaszyfrowany

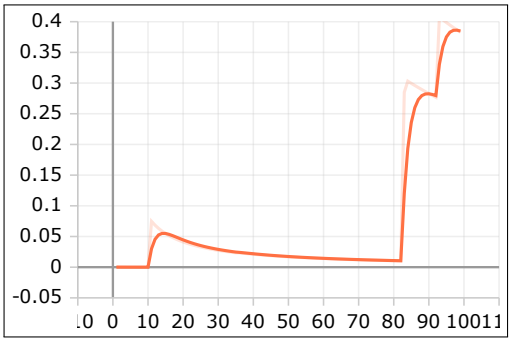
Wynik jest poprawny, a wartość predykcji modelu zaszyfrowanego ma niewielkie różnice z wartością predykcji modelu niezaszyfrowanego. Jest to spowodowane zaszumieniem, jednak nie wpływa ono na prawidłowy wynik.

3.1.4 Test zaszyfrowanego modelu liniowego i zaszyfrowanego modelu LeNet1.

Zaszyfrowano model liniowy, z funkcją aktywacyjną *Square*() i sprawdzono wynik jego precyzji na 100 zaszyfrowanych obrazach ze zbioru MNIST, wykres precyzji przedstawiono na grafice 3.8a, a wykres funkcji straty na grafice 3.8b. Średnią precyzję, czas, średnią wartość funkcji straty z tego testu przedstawiono w tabeli 3.8.



(a) Wykres precyzji



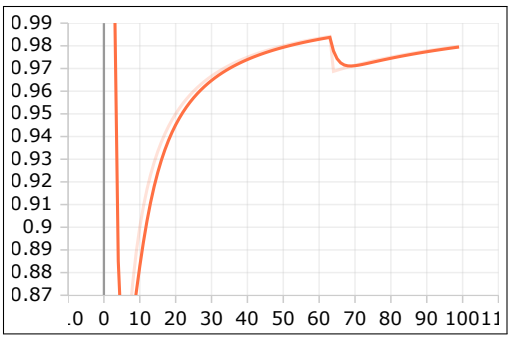
(b) Wykres funkcji straty

Rysunek 3.8: Test na 100 zaszyfrowanych obrazach ze zbioru MNIST, zaszyfrowanego liniowego modelu sieci korzystającego z funkcji *Square*().

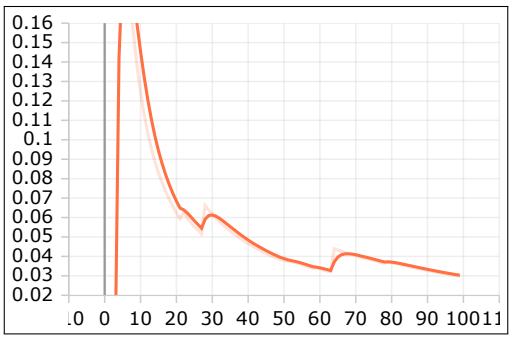
Czas	Precyzja	Strata	Model	Funkcja aktywacyjna
4h 24min 7s	96%	$3.8*(10)^{-1}$	Liniowy	Square()

Tablica 3.8: Tabela przedstawia wyniki uzyskane na 100 zaszyfrowanych obrazach ze zbioru MNIST podczas testu na zaszyfrowanym modelu liniowym korzystającego z funkcji aktywacyjnej *Square*().

Zaszyfrowano model LeNet1 z jedną funkcją aktywacyjną *Square*() i sprawdzono wynik jego precyzji na 100 zaszyfrowanych obrazach ze zbioru MNIST, wykres precyzji przedstawiono na grafice 3.9a, a wykres funkcji straty na grafice 3.9b. Średnią precyzję, czas, średnią wartość funkcji straty z tego testu przedstawiono w tabeli 3.9.



(a) Wykres precyzji



(b) Wykres funkcji straty

Rysunek 3.9: Test na 100 zaszyfrowanych obrazach ze zbioru MNIST, zaszyfrowanego modelu LeNet1 korzystającego z jednej funkcji aktywacyjnej *Square*().

Czas testu	Precyzja	Strata	Model	Funkcja aktywacyjna
4h 38min 30s	98%	$2.96*(10)^{-2}$	LeNet1	*Square*() użyta raz

Tablica 3.9: Tabela przedstawia wyniki uzyskane na 100 zaszyfrowanych obrazach ze zbioru MNIST podczas testu na zaszyfrowanym modelu LeNet1 korzystającego z jednej funkcji aktywacyjnej *Square*()

Modele zaszyfrowane osiągają zbliżone wyniki precyzji do wyników modeli nieszyfrowanych. Zaszumienie towarzyszące szyfrowaniu wpływa na różnice między wynikami, jednak są one na tyle małe, że nie powodują strat precyzji. Czasz działania modeli zaszyfrowanych jest dużo większy od czasu działania modeli niezaszyfrowanych.

3.1.5 Wnioski

Po wykonaniu eksperymentów i przeprowadzeniu analizy, nasuwają się następujące końcowe wnioski. Szyfrowanie homomorficzne pozwala na bezpieczne korzystanie z modelu i danych w środowiskach zewnętrzych, do których zalicza się chmura obliczeniowa. Dzięki zastosowaniu tego szyfrowania niewykonalne stają się: skopiowanie modelu przez osoby trzecie, poznanie danych na których sieć operuje, poznanie zwracanych przez model wyniki. W przypadku kradzieży całego modelu, który jest zaszyfrowany, bez posiadania klucza szyfrującego, niemożliwym będzie skorzystanie z sieci w poprawny sposób. Koszt wysokiego poziomu bezpieczeństwa to długi czas działania zasyfrowanej sieci neuronowej. W przypadkach potrzeby uzyskania natychmiastowych wyników, długi czas działania powoduje niekorzystność tego rozwiązania. Wraz ze wzrostem ilości warstw w sieci, czas przetwarzania obrazów wydłuża się.

3.2 Instalacja i wdrożenie

Zawartość została opisana w rozdziale A.
Aby uruchomić program, który posłużył do testowania modeli sieci neuronowych, należy uruchomić plik /Kod/engineer_thesis.ipynb. Plik można uruchomić w środowisku Google Colaboratory [1], wykorzystując funkcjonalność File→Upload nootebook i włączając odpowiednie komórki. Wyniki można przeglądać za pomocą biblioteki TensorBoard, która została wykorzytana przy wybranych grupach eksperymentów, nie bazujących na wyniku konsoli. Linie kodu, uruchamiające TensorBoard, są zaimplementowane w odpowiednich komórkach kodu.

Rozdział 4

Podsumowanie

W pracy, skonstruowano skrypt, który pozwolił nauczyć sieć neuronową określonego problemu klasyfikacji obrazów oraz skrypt pozwalający na zaszyfrowanie homomorficzne wyuczonej sieci neuronowej oraz zaszyfrowanie danych. Następnie przetestowano poprawność klasyfikacji zaszyfrowanych obrazów na zaszyfrowanych modelach sieci. Wygenerowano dane dla zadanych eksperymentów, a na ich podstawie porównano wyniki sieci szyfrowanych oraz nieszyfrowanych i przedstawiono wnioski.

Ponadto, po wykonaniu eksperymentów i przeprowadzeniu analizy, nasuwają się następujące końcowe wnioski. Szyfrowanie homomorficzne pozwala na bezpieczne korzystanie z modelu i danych w środowiskach zewnętrzych, do których zalicza się chmura obliczeniowa. Dzięki zastosowaniu tego szyfrowania niewykonalnym staje się skopiowanie modelu przez osoby trzecie oraz nie jest wykonalnym poznać dane na których sieć operuje, a także poznać zwracane przez model wyniki. Również w przypadku kradzieży całego modelu, który jest zaszyfrowany, bez posiadania klucza szyfrującego, niemożliwym będzie skorzystanie z niej. Jednak wysoki poziom bezpieczeństwa powoduje długi czas działania takiej sieci neuronowej, ze względu na sposób szyfrowania jakiemu poddawany jest model i dane, co w przypadkach potrzeby uzyskania natychmiastowych wyników wpływa na niekorzyść tego rozwiązania. Ponadto wraz ze wzrostem ilości warstw w sieci, czas przetwarzania obrazów wydłuża się.

Z powyższych wniosków wynikają kolejne hipotezy i propozycje dalszych badań. Przykładowo, w jaki sposób można zwiększyć szybkość działania modelu zaszyfrowanego homomorficznie. Jak odporne jest takie szyfrowanie na próbę złamania klucza szyfrującego przez komputer kwantowy. Jak przedstawiają się wyniki czasowe działania zaszyfrowanwgo modelu na danych o większym rozmiarze.

Bibliografia

[1] Google colaboratorys. https://colab.research.google.com/notebooks/intro.ipynb.

[2] Numpy. https://numpy.org/.

[3] Pyfhel. https://pyfhel.readthedocs.io/en/latest/.

[4] Python. https://www.python.org/.

[5] Pytorch. https://pytorch.org/.

[6] Tensorboard. https://www.tensorflow.org/tensorboard/.

[7] The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[8] J. Fan, F. Vercauteren. Somewhat practical fully homomorphic encryption. 2018.

[9] S. Hardy. A homomorphic encryption illustrated primer. 2018.

[10] H. K. Hossein Gholamalinezhad. Pooling methods in deep neural networks, a review.

[11] H. C. Martin Albrecht, Melissa Chase. Homomorphic encryption standard. 2018.

[12] L. D. Xiaoyong Yuan. Es attack: Model stealing against deep neural networks without data hurdles. 2020.

[13] L. B. Yann LeCun. Gradient-based learning applied to document recognition. 1998.

Załącznik A

Zawartość płyty CD

Płyta CD zawiera następujące pliki:

- folder ze zbiorem grafik wygenerowanych na potrzeby przeprowadzonych eksperymentów i dołączonych do tej pracy
- plik źródłowy z rozszerzeniem .ipynb, kod źródłowy napisany w chmurze Google Colab i zapisany jako notatnik Jupter Notebook. Kod posłużył do implementacji skryptu, który pozwolił zaimplementować sieć neuronową i wygenerować wyniki do przeprowadzanych eksperymentów.
- źródło latexowe dla pracy w formacie .tex
- praca inżynierska w formacie .pdf

Pliki zostały zamieszczone w folderach:

- **/Tekst/praca.tex**
- **/Tekst/praca.pdf**
- **/Tekst/resource/*** - wszystkie źródła dla praca.tex
- **/Kod/engineer_thesis.ipynb** - kod źródłowy