Rozdział 7

Nieświadome sieci neuronowe

Stanisław Barański[1]

[1] Katedra Architektury Systemów Komputerowych
Wydział Elektroniki, Telekomunikacji i Informatyki
Politechnika Gdańska
stanislaw.baranski@pg.edu.pl

Abstrakt

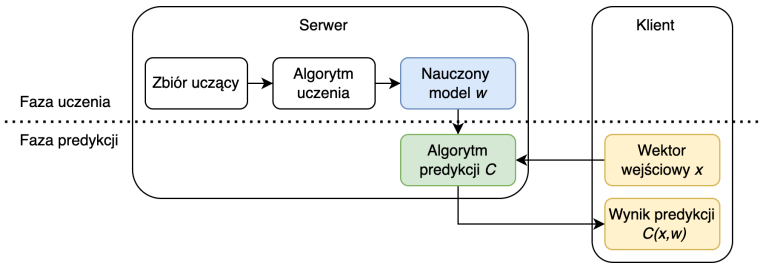
Coraz większą popularność zyskuje usługa predykcji za pomocą sieci neuronowych. Model ten zakłada istnienie serwera, który za pomocą wyuczonej sieci neuronowej dokonuje predykcji na danych otrzymanych od klienta. Model ten jest wygodny, ponieważ obie strony mogą skupić się na rozwoju w swojej specjalizacji. Wystawia on jednak na ryzyko utraty prywatności zarówno klienta, wysyłającego wrażliwe dane wejściowe, jak i serwer, udostępniający wyuczony model sieci neuronowej. Niniejszy rozdział opisuje proces dokonywania nieświadomej predykcji za pomocą sieci neuronowych. Nieświadomość pozwala na dokonanie predykcji za pomocą sieci neuronowych przy zachowaniu prywatności danych wejściowych klienta oraz modelu serwera. Umożliwia świadczenie usług, które dotychczas blokowane były przez regulacje prawne lub brak wiarygodności przetwarzających je systemów.

Słowa kluczowe: sieci neuronowe, nieświadome, prywatność, *multi-party computation*

7.1. Wstęp

Uczenie maszynowe (*machine learning*, ML) znajduje zastosowanie w wielu aplikacjach; najważniejsze z perspektywy niniejszego rozdziału to rozpoznawanie obrazów, diagnozy medyczne, ocena ryzyka udzielenia kredytu, i inne przetwarzające poufne dane osobiste.

Uczenie nadzorowane (rys. 7.1) składa się z dwóch faz: 1) faza uczenia, w której algorytm uczy model w zbiorem uczącym; 2) faza predykcji, w której algorytm predykcji C na podstawie danych wejściowych x (zwanych wektorem parametrów) oraz wcześniej wyuczonego modelu w dokonuje predykcji $C(x, w)$ wartości ciągłej (regresja) lub przypisania kategorii (klasyfikacja).



Rysunek 7.1: Diagram przedstawiający realizację usługi *Prediction as a service*

Szczególnie popularną metodą uczenia maszynowego, ze względu na ich wszechstronność, są sieci neuronowe (*neural networks*, NN).

Model biznesowy świadczenia predykcji za pomocą sieci neuronowych (*prediction as a service*) pozwala klientowi na oddelegowanie predykcji C danych wejściowych x do usługodawcy (dalej zwanego serwerem), który oferuje wytrenowany model sieci neuronowych w.

Model ten jest atrakcyjnym rozwiązaniem dla wielu biznesów, ponieważ usługodawcy mogą skupić się na optymalizacji modelu, a klienci zyskują przez ograniczenie kosztów potrzebnych na zatrudnienie specjalistów, zebranie danych treningowych, wytrenowanie modeli i utrzymanie infrastruktury.

Model ten jednak implikuje niewidoczne na pierwszy rzut oka trudności. W sytuacji, gdy dane wejściowe zawierają wrażliwe informacje, pojawiają się problemy związane z prywatnością i bezpieczeństwem przetwarzania danych (np. GDPR i HIPAA). Przykładem takich sytuacji może być laboratorium medyczne, które chciałoby dokonać rozpoznania choroby na podstawie zdjęć rentgenowskich lub oszacowania prawdopodobieństwa wystąpienia choroby na podstawie pobranych próbek. Regulacje dotyczące przetwarzania danych mogą uniemożliwić zarówno udostępnianie danych pacjenta do serwera, jak i udostępnianie przez serwer nauczonego modelu do laboratorium.

Innym przykładem może być klient, który chciałby otrzymać oszacowanie swojej zdolności kredytowej lub składek ubezpieczeniowych od instytucji finansowej. Zarówno klient nie chce przekazywać szczegółowych danych personalnych, jak i serwer nie chce udzielać szczegółów algorytmu estymacji.

W takich przypadkach oddelegowywanie predykcji staje się utrudnione, a nawet niemożliwe. Najprostszym rozwiązaniem tego problemu jest pobranie modelu w i dokonanie predykcji C lokalnie w domenie klienta. Rozwiązanie to jednak w wielu przypadkach jest niemożliwe ze względu na: 1) prawa własnościowe takich modeli; 2) utrudnioną aktualizację modelu; 3) możliwość ekstrakcji informacji o zbiorze treningowym, który również może być poufny, np. w przypadku historii danych medycznych pacjentów.

Nasuwa się więc pytanie, **czy możliwe jest dokonanie predykcji C tak, aby serwer nie dowiedział się niczego o danych wejściowych x, a klient nie dowiedział się niczego o modelu w w poza wynikiem predykcji $C(x, w)$?**

Jak się okazuje, jest to możliwe. Niniejszy rozdział prezentuje aktualny stan techniki realizacji tego typu rozwiązań. W dalszej części będziemy nazywać te techniki nieświadomymi (*oblivious*). Istnieją metody zapewniania nieświadomości wielu algorytmów uczenia maszynowego [5, 35], jak np. maszyny wektorów nośnych, naiwny klasyfikator bayesowski, Perceptron, AdaBoost. Jednak w tym rozdziale skupimy się na predykcji za pomocą nieświadomych sieci neuronowych. Pominiemy również fazę nieświadomego uczenia – zakładamy, że model został nauczony w dowolny sposób, a nieświadomość ogranicza się jedynie do fazy predykcji. W celu zrozumienia procesu unieświadamiania sieci neuronowych musimy najpierw ustalić, czym tak naprawdę jest sieć neuronowa. W sekcji 7.2 odpowiemy na to pytanie przez wizualizację oraz sformułowanie uniwersalnego modelu sieci neuronowych.

W sekcji 7.3 przejdziemy do tematu nieświadomych obliczeń. Wymagać to będzie zaczerpnięcia wiedzy z czterech dziedziny nauki: teorii obliczeń, teorii układów cyfrowych, kryptografii, oraz uczenia maszynowego.

Aby przekazać tak szeroką wiedzę w jednym rozdziale, skupimy się na jednym – najpopularniejszym i najprostszym – rozwiązaniu do budowania nieświadomych sieci neuronowych. Następnie, korzystając z nabytej intuicji, rozpatrzymy alternatywne podejścia (sekcja 7.5).

7.2. Sieci neuronowe

Sieć nieuronowa jest potokiem warstw. Każda warstwa otrzymuje sygnał wejściowy, przetwarza go i generuje sygnał wyjściowy, który staje się sygnałem wejściowym dla kolejnej warstwy. Pierwsza warstwa otrzymuje dane wejściowe x, a sygnał wyjściowy ostatniej warstwy jest wynikiem predykcji $C(x, w)$.

Typowa warstwa sieci neuronowej dokonuje transformacji liniowej (mnożenie i dodawanie macierzy), a następnie transformacji nieliniowej (funkcja aktywacji, a czasami również *pooling* w celu zredukowania rozdzielczości danych).

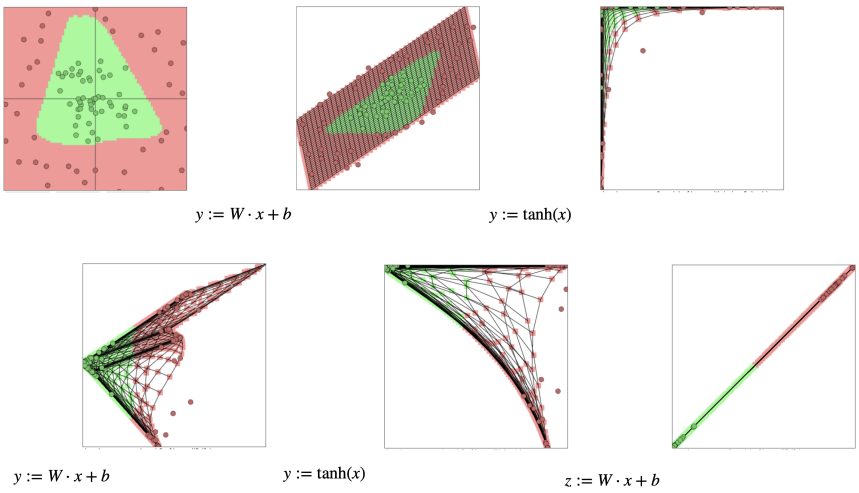
Predykcje za pomocą sieci neuronowych można przedstawić za pomocą potoku transformacji:

$$x \to f_1 \to a_1 \to ... \to f_n \to a_n \to y$$

gdzie:

- x jest wektorem wejściowym;
- f_i jest transformacją liniową warstwy i;
- a_i jest transformacją nieliniową warstwy i;
- n jest łączną liczbą warstw sieci neuronowej.

Celem transformacji jest zniekształcenie przestrzeni danych wejściowych w taki sposób, aby stały się liniowo separowalne, tj. aby dało się stworzyć linię (jeśli dane wejściowe opisane są w dwóch wymiarach), płaszczyznę (w trzech wymiarach), hiperpłaszczyznę (w $n + 1$ wymiarach), która podzieli zbiór wejściowy na dwa podzbiory. Rys. 7.2 przedstawia transformacje każdej z warstw przykładowej sieci neuronowej. Liniowa separowalność danych (na obszary zielony i czerwony) jest możliwa przez przeprowadzenie sekwencji transformacji liniowych i nieliniowych. Transformacje liniowe pozwalają na obracanie, przechylanie oraz rozciąganie przestrzeni. Transformacje nieliniowe natomiast pozwalają na deformacje przestrzeni.



Rysunek 7.2: Wizualizacja transformacji liniowych i nieliniowych w celu osiągnięcia liniowej separowalności

7.2.1. Transformacje liniowe

Mnożenie i dodawanie macierzy są najpopularniejszymi transformacjami liniowymi stosowanymi w sieciach neuronowych:

$$y := W \cdot x + b \tag{7.1}$$

gdzie:

- x jest wektorem wejściowym;
- W jest macierzą wag;
- b jest wektorem wyrazów wolnych;
- y jest wektorem wyjściowym.

Konwolucja jest transformacją liniową pozwalającą na obliczenie produktu skalarnego pomiędzy tensorem wag (filtra, zwanego kernelem) a elementem macierzy wraz z jego sąsiadującymi elementami. Proces ten jest powtarzany z przesuwaniem filtra po macierzy wejściowej. W praktyce konwolucje zamienia się na postać mnożenia i dodawania macierzy, osiągając wzrost wydajności [12], podobnie jak w równaniu (7.1), z tą różnicą, że dana wejściowa i wyraz wolny są macierzami: $Y := W \cdot X + B$.

7.2.2. Transformacje nieliniowe

Sieci neuronowe korzystają z nieliniowych transformacji w celu zamodelowania nieliniowej zależności pomiędzy przestrzeniami wejściową a wyjściową.
Funkcje aktywacji. Rozróżnić można trzy kategorie funkcji aktywacji.

- *Funkcja aktywacji przedziałami liniowymi.* Jest to funkcja, która może zostać zareprezentowana jako zbiór n funkcji liniowych $f_i(x) = a_i x + b_i$, gdzie x jest ograniczone przez zakresy dolny i górny dla danego przedziału. Funkcjami tego typu są:
  - funkcja tożsamościowa: $f(x) = x$;
  - *rectified linear units (ReLU)*: $f(x) = max(0, x)$;
  - *leaky ReLU*: $f(x) = max(0, x) + amin(0, x)$;
  - *maxout*: $f(x) = max(y_1, ..., y_n)$.
- *Gładka (regularna) funkcja aktywacji.* Jest to funkcja różniczkowalna określona pewną liczbę razy w swojej dziedzinie. Najpopularniejsze gładkie funkcje aktywacji to:
  - *sigmoid (logistic)*: $f(x) = \frac{1}{1+e^{-x}}$;
  - *hyperbolic tangent (tanh)*: $f(x) = \frac{e^{2x}-1}{e^{2x}+1}$;
  - *softplus*: $f(x) = \log(e^x + 1)$.

  Sigmoid oraz *tanh* są funkcjami zwanymi sigmoidalnymi, pomiędzy którymi zachodzi zależność:

$$tanh(x) = 2 \cdot sigmoid(2x) - 1 \tag{7.2}$$

- *Softmax.* Funkcja ta używana jest najczęściej jako ostatnia warstwa NN w celu określenia rozkładu prawdopodobieństwa klasyfikacji. Funkcja zdefiniowana jest jako

$$softmax_i(x) = \frac{e^{x_i}}{\sum_k e^{x_k}} \tag{7.3}$$

Pooling jest operacją redukującą rozdzielczość macierzy. Nazywana również operacją złożenia, polega na organizacji danych wejściowych w podgrupy i agregacji każdej podgrupy przy zmniejszaniu wymiaru danych wejściowych. Najczęściej elementy agreguje się za pomocą funkcji uśredniania (*mean pooling*) lub maksymalizacji (*max pooling*).

7.2.3. Wnioski

Faza predykcji za pomocą sieci neuronowych sprowadza się do sekwencji transformacji liniowych oraz tranformacji nieliniowych. Transformacje liniowe sprowadzają się do mnożenia i dodawania macierzy. Transformacje nieliniowe sprowadzają się do wykonania funkcji aktywacji lub operacji *poolingu*.

Dlatego, aby unieświadomić cały proces predykcji, wystarczy unieświadomić operacje mnożenia i dodawania macierzy oraz wykonywania funkcji aktywacji i operacji *poolingu*.

7.3. Nieświadome obliczenia

Secure multi-party computation (MPC) jest techniką wykonywania dowolnego algorytmu $F(x_1, ..., x_n)$ przez n niezależnych uczestników w taki sposób, że żaden z nich nie dowiaduje się niczego o danych wejściowych pozostałych uczestników.

Dla naszych potrzeb skupimy się na przypadku, w którym tylko dwie strony (klient i serwer) biorą udział w wykonaniu algorytmu F. Ten szczególny przypadek nazywany jest *secure two-party computation* (2PC).

Najprostszym sposobem na wykonanie algorytmu F w taki sposób, aby żadna ze stron nie dowiedziała się o danych wejściowych drugiej strony, jest wprowadzenie zaufanej trzeciej strony (*trusted third party*, TTP), która otrzymuje dane wejściowe od obu uczestników, wykonuje funkcjonalność F oraz zwraca wynik obliczeń.

Wprowadzenie zaufanej trzeciej strony jest problematyczne z wielu powodów, najważniejszym z nich jest wprowadzenie – często nieakceptowalnego – założenia o faktycznej uczciwości TTP. Dlatego jest to założenie, którego za wszelką cenę staramy się uniknąć, projektując zaufane systemy informatyczne.

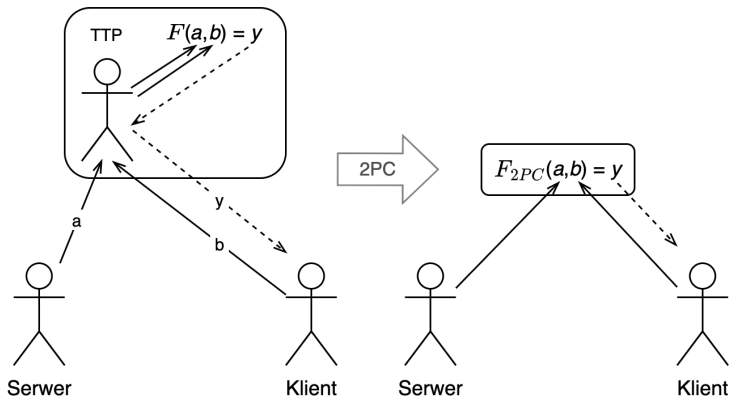
Technika 2PC rozwiązuje ten problem, pozwalając na osiągnięcie tej samej funkcjonalności bez pomocy TTP (rys. 7.3).

7.3.1. *Yao's garbled circuit* (GC)

Przez wiele lat 2PC stało w cieniu teoretycznych rozważań kryptologów. Pierwszym praktycznym rozwiązaniem jest *Yao's garbled circuit* (GC) [36].

Intuicja protokołu GC

W celu pokazania działania protokołu GC posłużymy się przykładem problemu dwóch studentów chcących dowiedzieć się, kto z nich jest bogatszy, bez ujaw-



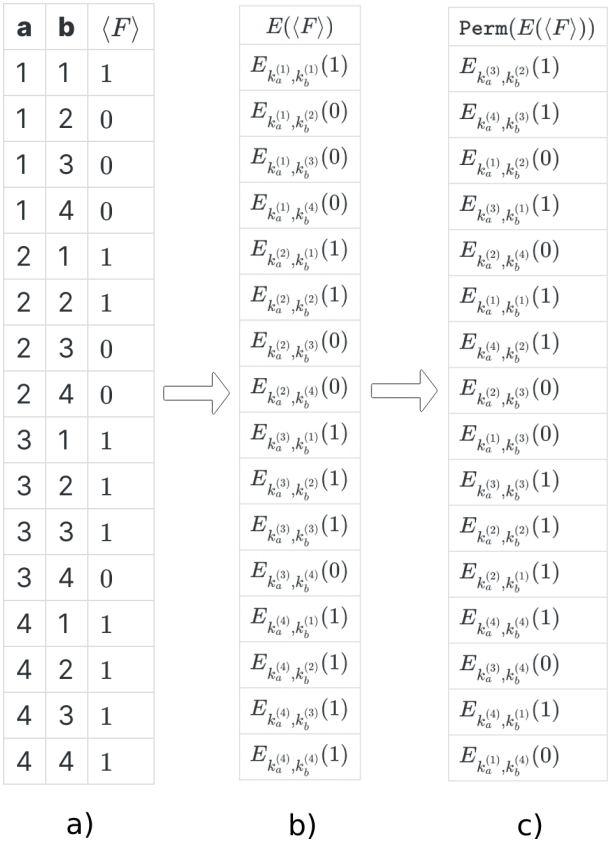
Rysunek 7.3: Transformacja bezpiecznej funkcjonalności F w modelu z zaufaną trzecią stroną (TTP) do bezpiecznej funkcjonalności bez zaufanej trzeciej strony F_{2PC} (*secure two-party computation*)

nienia swojego majątku. Problem ten można zareprezentować przez funkcję $F(a, b) = a \geq b$, gdzie a to majątek pierwszego studenta (dla spójności nazywanego serwerem), a b to majątek drugiego studenta (klienta). Dla uproszczenia przykładu załóżmy, że liczby a i b reprezentować będą liczbę cyfr majątku każdej ze stron oraz że maksymalna liczba cyfr to 4. Przykładowo, majątek o wartości 7 zł reprezentowany jest przez liczbę 1, majątek o wartości o wartości 35 zł przez liczbę 2, o wartości 786 zł przez liczbę 3, o wartości 9999 zł przez liczbę 4.

Dziedziną funkcji $F\colon X^2 \to Y$ jest $X^2 = \{1, 2, 3, 4\}^2$, a przeciwdziedziną $Y = \{1, 0\}$, gdzie 1 reprezentuje prawdę (pierwszy argument jest większy lub równy drugiemu), natomiast 0 fałsz (pierwszy argument jest mniejszy od drugiego). W pierwszym kroku jedna ze stron (załóżmy, że będzie to serwer) tworzy tablice (*lookup table*) wszystkich możliwych wywołań funkcji F, tj. $\langle F \rangle = \langle F(1,1) = 1, F(1,2) = 0, ..., F(4,4) = 1 \rangle$ (rys. 7.4a). Wywołanie funkcji reprezentowanej w formie tablicy sprowadza się do wyciągnięcia odpowiedniego wiersza z kolumny $\langle F \rangle$ dla wartości a i b.

W celu stworzenia nieświadomej wersji funkcji F reprezentowanej w formie tablicy $\langle F \rangle$ serwer szyfruje tablicę, używając losowo wygenerowanych kluczy dla każdej wartości a oraz b. Konkretniej, każdej wartości $a \in X$ oraz $b \in X$ serwer przypisuje silny pseudolosowy klucz $k_a \in \{0, 1\}^\kappa$, oraz $k_b \in \{0, 1\}^\kappa$, gdzie κ jest parametrem bezpieczeństwa i oznacza długość ciągu bitów klucza szyfrującego.

Powstaje w ten sposób łącznie $|X|+|X| = 8$ kluczy ($k_a^{(1)}$, $k_a^{(2)}$, $k_a^{(3)}$, $k_a^{(4)}$, $k_b^{(1)}$, $k_b^{(2)}$, $k_b^{(3)}$, $k_b^{(4)}$) służących do zaszyfrowania wyniku funkcji F dla każdej kombinacji argumentów a i b. Niech $E_{k_a,k_b}(\cdot)$ będzie symetryczną funkcją szyfrującą, która jest parametryzowana dwoma kluczami szyfrującymi: odpowiednim dla wartości a kluczem serwera k_a oraz odpowiednim dla wartości b kluczem klienta k_b. Szyfrowanie dwoma kluczami można implementować na wiele sposobów,



Rysunek 7.4: Transformacje tablicy protokołu *garbled circuit* (GC)

jednym z nich jest podwójne szyfrowanie, pierwszy raz kluczem serwera, drugi raz kluczem klienta; formalnie $E_{k_a,k_b}(\cdot) = E_{k_b}(E_{k_a}(\cdot))$.

Zaszyfrowaną tablicę $\langle F \rangle$ oznaczać będziemy $E(\langle F \rangle)$ (rys. 7.4b).

Tak przygotowana przez serwer tablica jest prawie gotowa do wysłania klientowi. Prawie, ponieważ otrzymując tablicę w takiej postaci, klient jest w stanie wydedukować a i b na podstawie wiersza tablicy. Dlatego ostatnią modyfikacją jest dokonanie losowej permutacji, tak aby klient nie był w stanie wydedukować wartości a. Powstaje w ten sposób tablica $\mathrm{Perm}(E(\langle F \rangle))$ widoczna na rys. 7.4c.

W tym momencie tablica jest gotowa. Serwer wysyła tablicę klientowi wraz z kluczem $k_a^{(i)}$, gdzie i jest wartością wybraną przez serwer; załóżmy, że wartość ta wynosi $i = 4$, dlatego serwer wysyła klientowi klucz $k_a^{(4)}$.

Zauważmy, że sam klucz $k_a^{(4)}$ jest losowym ciągiem bitów nienoszącym żadnej informacji. Nie pozwala na odszyfrowanie żadnego wiersza w tablicy. Nie pozwala również na określenie danej wejściowej serwera.

Nieświadomość obliczenia osiągana jest dopiero w następnym kroku, a przebiega on następująco. Serwer przeprowadza z klientem procedurę nieświadomego transferu (*oblivious transfer*, OT) [28], która pozwala klientowi wybrać jeden klucz ze zbioru oferowanych kluczy $\{k_b^{(1)}, k_b^{(2)}, k_b^{(3)}, k_b^{(4)}\}$ w taki sposób, że serwer nie dowiaduje się, który element został wybrany, a klient nie dowiaduje się niczego o pozostałych kluczach poza tym, który wybrał. Dla naszego przykładu załóżmy, że dla klienta liczba cyfr jego majątku wynosi $j = 3$, dlatego pobiera on klucz $k_b^{(3)}$.

W tym momencie klient posiada wszystkie elementy potrzebne do odtworzenia wartości funkcji $F(a, b)$. W połączeniu z otrzymanym kluczem $k_a^{(4)}$ klient może odszyfrować tylko i wyłącznie wiersz zaszyfrowany przy użyciu obu kluczy $k_a^{(4)}$ oraz $k_b^{(3)}$, czyli wiersz drugi o wartości $E_{k_a^{(4)},k_b^{(3)}}(1)$. Klient odszyfrowuje wiersz $E_{k_a^{(4)},k_b^{(3)}}(E_{k_a^{(4)},k_b^{(3)}}(1)) = 1$, otrzymując wartość pozytywną, mówiącą o większym majątku serwera $F(4, 3) = 4 \geq 3$. W zależności od uzgodnienia klient wysyła (lub nie) wynik do serwera. W celu zapewnienia wiarygodności wyniku zwróconego przez klienta wynik powinien zostać opatrzony podpisem cyfrowym wykonanym przez serwer.

Właściwy protokół GC

Uważny czytelnik może dostrzec dwa problemy wyżej opisanego protokołu. Po pierwsze, klient nie ma informacji, który wiersz odpowiada której wartości, dlatego nie wie, który wiersz powinien odszyfrować, używając otrzymanych kluczy. Naiwną metodą jest dodanie do każdej wartości wynikowej ciągu zer, które będą sygnalizowały, że to, co odszyfrował, faktycznie jest wartością wynikową, a nie losowym ciągiem bitów. Prawdopodobieństwo odszyfrowania błędnego wiersza zakończonego ciągiem zer jest bardzo niskie ($p = \frac{1}{2^\sigma}$, gdzie σ to liczba zer). Rozwiązanie to nie jest idealne, ponieważ klient musi dokonać średnio $\frac{\langle F \rangle}{2}$ odszyfrowań, zanim dotrze do poprawnego wiersza w tablicy. W praktyce stosuje się technikę *point-and-permute* [3], która pozwala na dodanie wskaźnika lokalizującego wiersz do odszyfrowania. Metoda została opisana w dalszej części tej sekcji.

Drugim problemem jest rozmiar tablicy, który rośnie kwadratowo wraz z rozmiarem argumentów $|\langle F \rangle| = |X| \times |X| = |X|^2$. Gdybyśmy chcieli zmienić dziedzinę funkcji F na liczby typu uint32, rozmiar tablicy wyniósłby $|\text{uint32}| \times |\text{uint32}| = 2^{32} \times 2^{32} = 2^{64}$, następnie każdą liczbę należałoby zakodować na 32 bitach, w wyniku czego rozmiar tablicy wyniósłby $2^{64} \times 32$ bitów, czyli zdecydowanie za dużo, jak na proste porównanie dwóch liczb.

Niemniej dla małych wartości, jak np. bramki logiczne, dla których dziedzina wynosi $|\{0, 1\}| \times |\{0, 1\}| = 4$, rozwiązanie to jest praktyczne. Dlatego **protokół GC interpretuje algorytm F w postaci układu logicznego** C, składającego się ze zbioru bramek logicznych $G = g_1, g_2, ..., g_m$, połączonych przewodami $W = w_1, w_2, ..., w_n$. Każda bramka g_i jest funkcją przyjmującą wartości dwóch przewodów wejściowych i zwracającą wartość przewodu wyjściowego. Przykładowo niech g_{OR} będzie bramką OR, której przewodami wejściowymi są w_1

oraz w_2, a przewodem wyjściowym jest w_3. Bramka g_{OR} reprezentuje funkcję $w_3 \leftarrow g_{\text{OR}}(w_1, w_2)$, taką że:

$$0 \leftarrow g_{\text{OR}}(0,0),\ 1 \leftarrow g_{\text{OR}}(0,1),\ 1 \leftarrow g_{\text{OR}}(1,0),\ 1 \leftarrow g_{\text{OR}}(1,1)$$

Z teorii obliczeń wiemy, że każdy algorytm opisany w modelu maszyny RAM (random-access machine) może zostać zareprezentowany w postaci sieci bramek logicznych i wzajemnie (dowód polega na emulacji jednego modelu w drugim w czasie wielomianowym) [15]. W praktyce wystarczy zastosowanie trzech rodzajów bramek: NOT, XOR, AND, które w ciele skończonym $\mathbb{Z}_2$ (arytmetyka binarna) odpowiadają za negacje, dodawanie i mnożenie, pozwalając na reprezentację dowolnej innej bramki logicznej – są funkcjonalnie skończone. Co więcej, zapewniają algorytmiczną skończoność Turinga (*Turing completeness*), zatem pozwalają na implementację dowolnego algorytmu obliczeniowego.

Tak jak poprzednio, serwer szyfruje każdą z tablic prawdy losowo wygenerowanymi kluczami, z tą różnicą, że do każdej wartości w tablicy dodany będzie klucz umożliwiający odszyfrowanie dalej zagnieżdżonych bramek. Ewaluacja układu przebiega podobnie jak w poprzednio opisanym przykładzie – za pomocą nieświadomego transferu (OT) serwer oferuje parę kluczy $(k_w^{(0)}, k_w^{(1)})$, z której klient wybiera klucz odpowiadający bitowi wejściowemu dla każdego przewodu $w \in W$ będącego wejściem układu C.

Konstrukcja zaszyfrowanego układu (GC) przebiega następująco:

1. Serwer generuje dla każdego przewodu $w \in W$ parę losowo wygenerowanych kluczy $(k_w^{(0)}, k_w^{(1)})$, reprezentujących semantyczną wartość 0 i 1.

2. Serwer generuje dla każdego przewodu $w \in W$ jednobitowy klucz $p_w \in \{0,1\}$, który posłuży zaszyfrowaniu faktycznej wartości przewodu. Jeśli wartość w tablicy prawdy bramki wynosi v, to po zaszyfrowaniu wynosić będzie $e_w = v_w \oplus p_w$, gdzie $\oplus$ jest operatorem XOR[1]. Szyfrowanie to przeciwdziała dedukcji wartości wejściowej drugiej strony na podstawie wartości wyjściowej. Przykładowo wiedząc, że na wyjściu bramki OR jest zero, wiemy, że na wyjściu musiały być dwa zera.

3. Dla każdej bramki w układzie $g \in G$ serwer generuje zaszyfrowaną tablicę prawdy. Wykorzystuje do tego przygotowane w poprzednich krokach zaszyfrowane wartości przewodów $e_w = v_w \oplus p_w$ oraz klucze $(k_w^{(0)}, k_w^{(1)})$ reprezentujące semantyczne wartości 0 i 1 przewodu $w \in W$. Zaszyfrowana tablica prawdy T_g dla każdej z bramek $g \in G$ ma następującą postać:

$$T_g = \begin{pmatrix} E_{k_m^{(0\oplus p_m)}, k_n^{(0\oplus p_n)}}(k_o^{(g(0\oplus p_m, 0\oplus p_n)}) || (g(0\oplus p_m, 0\oplus p_n) \oplus p_o) \\ E_{k_m^{(0\oplus p_m)}, k_n^{(1\oplus p_n)}}(k_o^{(g(0\oplus p_m, 1\oplus p_n)}) || (g(0\oplus p_m, 1\oplus p_n) \oplus p_o) \\ E_{k_m^{(1\oplus p_m)}, k_n^{(0\oplus p_n)}}(k_o^{(g(1\oplus p_m, 0\oplus p_n)}) || (g(1\oplus p_m, 0\oplus p_n) \oplus p_o) \\ E_{k_m^{(1\oplus p_m)}, k_n^{(1\oplus p_n)}}(k_o^{(g(1\oplus p_m, 1\oplus p_n)}) || (g(1\oplus p_m, 1\oplus p_n) \oplus p_o) \end{pmatrix}$$

[1]Operator XOR może zostać użyty jako (idealna) funkcja szyfrująca. Jest to możliwe dzięki właściwości $(m \oplus k) \oplus k = m$, gdzie m jest wiadomością, a k kluczem szyfrującym.

gdzie m i n są przewodami wejściowymi, o jest przewodem wyjściowym, p_w jest bitem szyfrującym wartość przewodu $w \in W$, a $||$ jest operatorem konkatenacji.

W ten sposób zaszyfrowane tablice zostają wysłane do klienta w celu **wykonania układu**, który przebiega następująco:

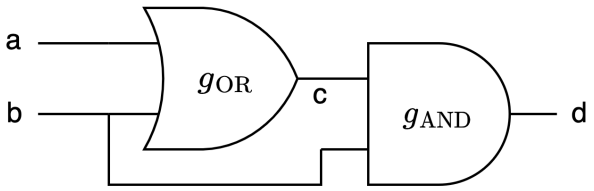
1. Serwer wysyła do klienta:
   - zaszyfrowane tablice T_g dla wszystkich bramek $g \in G$;
   - klucze szyfrujące $k_w^{(x)}$ dla każdego przewodu $w \in W$, które semantycznie odpowiadają danym wejściowym serwera, tj. jeśli dla przewodu i serwer wybrał wartość 0, wysyła on klucz $k_i^{(0 \oplus p_i)}$;
   - za pomocą nieświadomego transferu (OT) klucze szyfrujące $k_w^{(x)}$ dla każdego przewodu $w \in W$, które semantycznie odpowiadają danym wejściowym klienta;
   - bity p szyfrujące wartości przewodów wyjściowych układu.
2. Klient, korzystając z otrzymanych kluczy, dokonuje sekwencji rozszyfrowywania kolejnych bramek, a dokładniej, odpowiadających im zaszyfrowanych tablic prawdy. Ponieważ nie posiada on bitów szyfrujących p dla przewodów innych niż wyjściowe, nie może on wydedukować faktycznych wartości bramek.
3. Po wykonaniu sekwencji deszyfracji całego układu i otrzymaniu zaszyfrowanych wartości przewodów wyjściowych układu klient, korzystając z otrzymanych w pierwszym kroku kluczy p, odszyfrowuje wartości przewodów wyjściowych, ucząc się wartości wynikowej wykonanego układu.
4. W zależności od uzgodnienia klient wysyła (lub nie) wynik do serwera. W celu zapewnienia wiarygodności wyniku zwróconego przez klienta wynik bramek wyjściowych powinien zostać opatrzony podpisem cyfrowym wykonanym przez serwer.

Przykład protokołu GC

Weźmy dla przykładu funkcję $F(a, b) = (a \vee b) \wedge b$, gdzie a i b są wartościami logicznymi. Przewód a jest wejściem serwera, przewód b wejściem klienta, c jest przewodem pośrednim, a d jest przewodem wyjścia układu. Rys. 7.5 prezentuje układ C_F realizujący funkcję F. Układ składa się z przewodów $W = \{a, b, c, d\}$ oraz bramek $G = \{g_{\text{OR}}, g_{\text{AND}}\}$.

Konstrukcja zaszyfrowanego układu (GC) widocznego na rys. 7.6 przebiega następująco:

1. Serwer losowo generuje klucze szyfrujące wartości p_w dla każdego przewodu $w \in W$. Przyjmijmy $p_a = 1, p_b = 0, p_c = 1, p_d = 0$.



Rysunek 7.5: Układ C_F realizujący $F(a, b) = (a \vee b) \wedge b$

2. Serwer losowo generuje parę kluczy $(k_w^{(0)}, k_w^{(1)})$ odpowiadających semantycznie wartościom 0 i 1 dla każdego przewodu $w \in W$ oraz konkatenuje je z zaszyfrowaną (kluczem p_w) odpowiadającą wartością. Powstaje w ten sposób para $(k_w^{(0)}||(0 \oplus p_w), k_w^{(1)}||(1 \oplus p_w))$. Przykładowo dla przewodu a, gdzie klucz szyfrujący $p_a = 1$, para kluczy to $(k_a^{(0)}||(0 \oplus 1 = 1), k_a^{(1)}||(1 \oplus 1 = 0))$. Powstają w ten sposób pary kluczy:

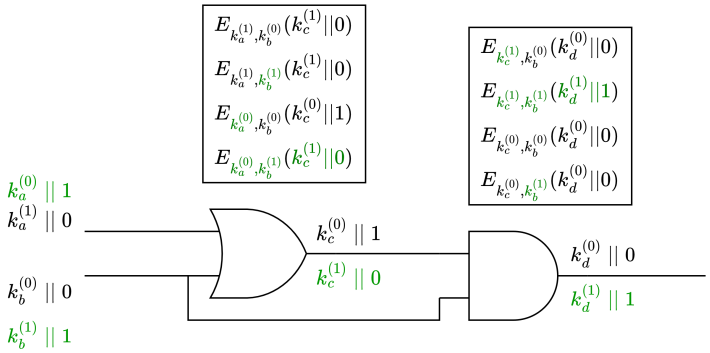
$$
\begin{gathered}
(k_a^{(0)}||1, k_a^{(1)}||0) \\
(k_b^{(0)}||0, k_b^{(1)}||1) \\
(k_c^{(0)}||1, k_c^{(1)}||0) \\
(k_d^{(0)}||0, k_d^{(1)}||1)
\end{gathered}
$$

3. Korzystając z kluczy k oraz p, serwer tworzy tablice prawdy dla każdej bramki. Przykładowo dla pierwszej bramki $T_{g_{\text{OR}}}$, gdzie przewodami wejściowymi są a i b, a wyjściowym jest c, tablica wygląda następująco:

$$
\begin{aligned}
T_{g_{\text{OR}}} &= \begin{pmatrix}
E_{k_a^{(0\oplus p_a)}, k_b^{(0\oplus p_b)}}(k_c^{(g(0\oplus p_a, 0\oplus p_b)}||(g(0 \oplus p_a, 0 \oplus p_b) \oplus p_c) \\
E_{k_a^{(0\oplus p_a)}, k_b^{(1\oplus p_b)}}(k_c^{(g(0\oplus p_a, 1\oplus p_b)}||(g(0 \oplus p_a, 1 \oplus p_b) \oplus p_c) \\
E_{k_a^{(1\oplus p_a)}, k_b^{(0\oplus p_b)}}(k_c^{(g(1\oplus p_a, 0\oplus p_b)}||(g(1 \oplus p_a, 0 \oplus p_b) \oplus p_c) \\
E_{k_a^{(1\oplus p_a)}, k_b^{(1\oplus p_b)}}(k_c^{(g(1\oplus p_a, 1\oplus p_b)}||(g(1 \oplus p_a, 1 \oplus p_b) \oplus p_c)
\end{pmatrix} \\
&= \begin{pmatrix}
E_{k_a^{(1)}, k_b^{(0)}}(k_c^{(1)}||0) \\
E_{k_a^{(1)}, k_b^{(1)}}(k_c^{(1)}||0) \\
E_{k_a^{(0)}, k_b^{(0)}}(k_c^{(0)}||1) \\
E_{k_a^{(0)}, k_b^{(1)}}(k_c^{(1)}||0)
\end{pmatrix}
\end{aligned}
$$

Gdy zaszyfrowany układ jest gotowy, możemy przejść do **fazy wykonania** (zwanej również fazą online). Przyjmijmy, że wartością wejściową serwera jest $a = 0$, klienta $b = 1$. Spodziewana wartość $F(0, 1) = (0 \vee 1) \wedge 1 = 1$.

Wykonanie układu przebiega następująco:



Rysunek 7.6: Zaszyfrowany układ (GC)

- Klient otrzymuje od serwera zaszyfrowane tablice $T_{g_{\text{OR}}}, T_{g_{\text{AND}}}$, klucz szyfrujący wyjście p_d oraz klucz odpowiadający semantycznie wartości wejściowej serwera. Serwer wybiera wartość 0 i przesyła odpowiadający klucz $k_a^0||1$. Klient widzi klucz k_a^0, który jest dla niego losowym ciągiem bitów, oraz zaszyfrowaną wartość 1, z której (bez klucza p_a) nie jest w stanie dowiedzieć się, czy faktyczna wartość to 0 czy 1.

- Korzystając z nieświadomego transferu, serwer oferuje klientowi parę kluczy $(k_b^{(0)}||0, k_b^{(1)}||1)$. Klient wybiera wartość 1 i otrzymuje klucz $k_b^1||1$.

- Posiadając wszystkie klucze wejściowe (k_a^0, $k_b^1||1$, oraz $p_d = 0$), klient rozpoczyna wykonywanie układu przez rozszyfrowywanie tablic. Klient wybiera wartość z tablicy na podstawie skonkatenowanych (uzewnętrznionych) zaszyfrowanych wartości przewodów wejściowych $e = v \oplus p$. Dla pierwszej bramki g_{OR} wartość uzewnętrzniona przewodu a wynosi 1, tak samo dla przewodu b uzewnętrzniona wartość wynosi 1, dlatego klient odszyfrowuje czwarty rząd tablicy $T_{g_{\text{OR}}}$. Na rys. 7.6 kolorem zielonym zaznaczone zostały wartości, których nauczył się klient. Wyjściem układu jest przewód d, którego wartość wynosi $k_d^{(1)}||1$. Korzystając z klucza $p_d = 0$, klient rozszyfrowuje ostateczną wartość $1 \oplus p_d = 1 \oplus 0 = 1$, która zgadza się z oczekiwaną wartością $F(0,1) = (0 \vee 1) \wedge 1 = 1$.

7.3.2. Kompilatory MPC

Funkcjonalność F, którą chcemy wykonać nieświadomie, musi zostać zamieniona na postać układu logicznego C_F, a dokładniej listy sieci (*netlist*), która następnie zostaje poddana unieświadomieniu przy pomocy protokołu GC.

Transformacja funkcjonalności F do postaci listy sieci jest metodą powszechnie stosowaną w projektowaniu układów cyfrowych. Można ją wykonywać natywnie, korzystając z języków takich jak np. Verilog, VHDL lub z syntezatorów

HLS (*high-level synthesis*) pozwalających na syntezę z języków wysokiego poziomu, np. z języka C (SPARK [18], CBMC-GC [21]) lub Python (MyHDL[2]). W rezultacie powstaje sprzętowy opis układu logicznego w formacie HDL (*hardware description language*), który pozwala na stworzenie uproszczonej listy sieci opisującej elementy układu i połączenia między nimi.

Następnie tak utworzona lista sieci może zostać uruchomiona nieświadomie przy pomocy protokołu GC.

Istnieją narzędzia umożliwiające dokonywanie wyżej wymienionych transformacji. Część z nich przyjmuje za dane wejściowe sprzętowy opis układ logicznego HDL, przykładowo: TinyGarble [33][3], ABY [13][4]. Inne dostarczają pełny zestaw narzędzi, od języka programowania do środowiska uruchomieniowego, przykładami są: MP-SPDZ [23][5], MPyC [32][6], EzPC [11][7], EMP-Toolkit [34][8].

Analiza porównawcza kompilatorów MPC dostępna jest w przeglądzie [19][9].

7.4. Nieświadome sieci neuronowe

Z sekcji 7.3 wiemy, jak zamienić dowolny algorytm na algorytm w wersji nieświadomej, korzystając z protokołu GC. Z sekcji 7.2 wiemy również, jak dokonywana jest predykcja przy użyciu sieci neuronowych. Pozostaje połączyć zdobytą wiedzę do stworzenia nieświadomej predykcji sieci neuronowej.

Predykcja przy użyciu NN może zostać zamodelowana jako rekurencyjna sekwencja mnożenia i dodawania macierzy oraz funkcji aktywacji/*poolingu*:

$$z := W_L * f_{L-1}(...f_1(W_1 * X + B_1)...) + B_L$$

gdzie:

- $W_1, W_2, ..., W_L$ są macierzami wag sygnałów pojawiających się na wejściach poszczególnych neuronów każdej z L warstw;
- $B_1, B_2, ..., B_L$ są wektorami wyrazów wolnych (*bias*) dla każdej z L warstw;
- $f_1, ..., f_{L-1}$ są funkcjami aktywacji/*poolingu* dla każdej z L warstw;
- X jest wektorem wejściowym;
- z jest wynikiem predykcji.

Celem nieświadomych sieci neuronowych jest obliczenie wartości z w taki sposób, aby klient nie dowiedział się niczego o $(W_1, W_2, ..., W_L)$ oraz $(B_1, B_2, ..., B_L)$, a serwer nie dowiedział się niczego o X oraz z.

[2]https://www.myhdl.org/
[3]https://github.com/esonghori/TinyGarble
[4]https://github.com/encryptogroup/ABY
[5]https://github.com/data61/MP-SPDZ
[6]https://github.com/lschoe/mpyc/
[7]https://github.com/mpc-msri/EzPC
[8]https://github.com/emp-toolkit
[9]https://github.com/MPC-SoK/frameworks

Aby osiągnąć taką izolację, podstawiamy dane wejściowe każdej ze stron pod model funkcjonalności $F_{2PC}(a, b)$, gdzie a jest prywatną daną wejściową jednej ze stron, a b prywatną daną wejściową drugiej ze stron. W naszym przypadku $a = X$ $b = ((W_1, W_2, ..., W_L), (B_1, B_2, ..., B_L))$. W wyniku powstaje:

$$F_{2PC}(X, ((W_1, W_2, ..., W_L), (B_1, B_2, ..., B_L))) \\ = W_L * f_{L-1}(...f_1(W_1 * X + B_1)...) + B_L$$

Funkcje aktywacji $f_1, ..., f_{L-1}$ oraz rozmiary macierzy są częścią znanego obu stronom układu logicznego, który zostaje wykonany wspólnie przez obie strony.

Tak zareprezentowana predykcja jako dwuargumentowa funkcja F zostaje zamieniona na postać listy sieci, a następnie wykonana wspólnie przez obie strony przy wykorzystaniu protokołu GC opisanego w sekcji 7.3.

Finalny przebieg protokołu został zwizualizowany na rys. 7.7 i wygląda następująco:

1. Serwer dokonuje syntezy architektury modelu sieci neuronowej do postaci netlisty, którą przesyła klientowi.
2. Klient tworzy, postępując zgodnie z protokołem GC, zaszyfrowany układ (*garbled circuit*), a dokładniej zaszyfrowane tablice (*garbled tables*) i przesyła je do serwera.
3. Klient przesyła klucze reprezentujące jego dane wejściowe oraz oferuje serwerowi klucze odpowiadające bitom reprezentującym wyuczone wagi modelu.
4. Serwer, korzystając z otrzymanych kluczy, realizuje układ (odszyfrowując kolejne tablice), a następnie wysyła zaszyfrowany wynik klientowi.
5. Klient odszyfrowuje wynik, otrzymując predykcję.

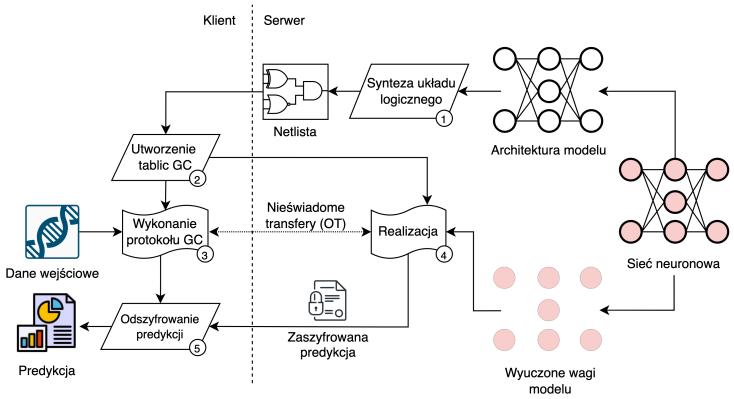
Nieświadome transformacje liniowe

Jak ustaliliśmy w sekcji 7.2.3, transformacje liniowe sprowadzają się do mnożenia i dodawania macierzy.

Mnożenie i dodawanie macierzy polegają na wykonaniu operacji dodawania i mnożenia wag, wyrazów wolnych oraz danych wejściowych.

Nieświadome transformacje nieliniowe

Funkcja aktywacji przedziałami liniowa jest prosta do unieświadomienia. Przykładowo ReLU polega na skonstruowaniu układu, który porówna y oraz 0 i na podstawie wyniku zwróci y lub 0. Wszystkie funkcje aktywacji przedziałami liniowe są możliwe do zrealizowania operacjami odejmowania, dodawania i mnożenia.



Rysunek 7.7: Przebieg nieświadomej predykcji sieci neuronowych

Gładka funkcja aktywacji jest trudniejsza do unieświadomienia, ponieważ wymaga skomplikowanych operacji dzielenia oraz potęgowania przez zmienną. Dlatego w praktyce gładkie funkcje zamienia się na przybliżone funkcje przedziałami liniowe. Dzięki temu zyskuje się wydajniejsze i prostsze w kompilacji funkcje aktywacji, kosztem nieznacznego spadku dokładności predykcji.

Pooling sprowadza się do wielokrotnego wykonania operacji porównania (*max pooling*) lub sumowania (*mean pooling*) na elementach agregowanego regionu, które również są proste w implementacji.

7.5. Stan techniki

Opisane w tym rozdziale podejście do wykonywania nieświadomych predykcji bazuje na rozwiązaniu **DeepSecure** [31] (DAC 2018), które w całości bazuje na protokole GC. Istnieje jednak wiele innych podejść korzystających z innych protokołów kryptograficznych do osiągania MPC.

Homomorficzne szyfrowanie (*homomorphic encryption*, HE) pozwala na wykonywanie obliczeń na zaszyfrowanych danych. Istnieją dwie wersje HE: pełne (*fully homomorphic encryption*, FHE) [9] oraz niepełne (*partially homomorphic encryption*, PHE) [8, 27]. FHE pozwala na wykonywanie obliczeń na zaszyfrowanych danych, jednak przy bardzo dużym narzucie obliczeniowym, co wyklucza je z większości praktycznych zastosowań. PHE natomiast ma dużo mniejszy narzut obliczeniowy, jednak pozwala na wykonywanie tylko podzbioru operacji (dodawanie lub mnożenie) lub wymaga ograniczenia głębokości układu reprezentującego algorytm. Co gorsza, funkcje nieliniowe, takie jak ReLU, nie są wspierane przez HE [30].

Współdzielenie sekretu (*secret sharing*, SS) pozwala na dokonywanie obliczeń w modelu MPC przy stosunkowo niskim narzucie obliczeniowym. SS polega na potraktowaniu danych wejściowych każdej ze stron jako sekret, który następnie zostaje podzielony pomiędzy każdą ze stron. Korzystając z właściwości SS, części sekretu mogą być poddawane operacjom arytmetycznym, a następnie zrekonstruowane, co prowadzi do uzyskania wyniku obliczeń [14]. Pierwszy protokół MPC korzystający z tej właściwości to **GMW** [17]. Pozwala on na wykonanie operacji NOT oraz XOR bez wzajemnej interakcji pomiędzy stronami; bramki AND – podobnie jak w przypadku GC – wymagają wykonania nieświadomego transferu (OT). Rozwinięciem protokołu GMW jest **BGW** [4] bazujący na schemacie współdzielenia sekretu Shamira (*Shamir secret sharing*, SSS), a dokładniej na homomorficzności wielomianów. Reprezentacja danych w postaci wielomianów pozwala na przedstawienie funkcjonalności F w postaci układu arytmetycznego, który umożliwia bardziej kompaktową reprezentację algorytmu niż układ logiczny. Przy wykorzystaniu homomorficzności wielomianów zarówno dodawanie, jak i mnożenie może zostać wykonane lokalnie – bez konieczności komunikacji między stronami. Obliczenia przy użyciu homomorficzności wielomianów w schemacie Shamira mają jednak jedną wadę. Mnożenie dwóch wielomianów stopnia t powoduje dwukrotny wzrost stopnia wielomianu, co z kolei sprawia, że rekonstrukcja sekretu wymaga użycia dwukrotnie większej liczby części sekretu. Powoduje to konieczność stosowania skomplikowanej procedury redukcji stopnia wielomianu oraz wymaga, aby co najmniej $2t + 1 = 3$ niezależne strony (brak większościowej koalicji) brały udział w protokole, a co za tym idzie, stworzenie 2PC nie jest możliwe (wymagane są co najmniej trzy strony). Z tego powodu SS często nie wspiera operacji mnożenia, a jedynie operację dodawania. Tak zawężony w swej funkcjonalności schemat nazywany jest addytywnym dzieleniem sekretu (*additive secret sharing*, ASS).

Trójki Bravera. Rozwiązaniem powyższych problemów jest zastosowanie – przełomowych w obszarze MPC – trójek Bravera [2], które pozwalają na dokonywanie operacji mnożenia w prostszy sposób. Trójki Bravera wymagają podziału protokołu na dwie fazy: i) jednorazowa faza offline, ii) każdorazowa faza online.

- W **fazie offline** dla każdej bramki mnożenia generowana jest trójka liczb (a, b, c) takich, że a i b są losowymi wartościami, a $c = ab$. Każda z liczb generowana jest w częściach, w taki sposób, aby żadna ze stron protokołu nie miała wglądu do pełnej wartości liczby, a jedynie do swojej części. Dopiero współpraca wszystkich stron protokołu i zebranie wszystkich części pozwalają na rekonstrukcję właściwej liczby. Część liczby x oznaczana jest symbolem $[x]$. Każda ze stron posiada zestaw części każdej z liczb, tj. $[a], [b], [c]$. Do wygenerowania współdzielonych trójek Bravera można wykorzystać protokół GMW [14] lub PHE [25].

- W **fazie online**, gdy w układzie pojawia się bramka odpowiedzialna za mnożenie dwóch wartości α i β, zostają one współdzielone przez obie strony jako wartości $[\alpha]$ oraz $[\beta]$. Następnie każda ze stron, używając swo-

ich części, liczy zamaskowaną wartość $[d] = [\alpha] - [a]$ oraz $[e] = [\beta] - [b]$, a następnie wysyła pozostałej stronie $[d]$ oraz $[e]$.

Jako że wartość jest zamaskowana wartością, nie niesie ona informacji o faktycznej wartości. Po otrzymaniu pozostałej części wartości d oraz e możliwa jest ich rekonstrukcja. Następnie, korzystając z równoważności:

$$\begin{aligned} & de + db + ae + c \\ & = de + db + ae + ab \\ & = (d + a)(e + b) \\ & = (\alpha - a + a)(\beta - b + b) \\ & = \alpha\beta \end{aligned}$$

każda ze stron liczy lokalnie część wyniku faktycznego mnożenia $[\alpha\beta] = de + d[b] + e[a] + [c]$.

Optymalizacja polega na tym, że dodawanie oraz mnożenie współdzielonej wartości $[x]$ przez dowolną stałą (która nie jest współdzielona) są proste i mogą zostać przeprowadzone lokalnie, bez konieczności wykonywania skomplikowanej procedury nieświadomego transferu i redukcji stopnia wielomianu [14].

MiniONN [25] (CCS 2017) w porównaniu z DeepSecure jest rozwiązaniem bardziej złożonym, ponieważ korzysta ze wszystkich wyżej wymienionych kryptosystemów: GC, HE, ASS oraz trójek Beavera. Kompilacja układu dokonywana jest za pomocą narzędzia ABY [13]. Mimo narzutów związanych z konwersjami pomiędzy kryptosystemami takie rozwiązanie (nazywane hybrydowym) osiąga dużo większą wydajność.

Chameleon [29] (AsiaCCS 2018), podobnie jak MiniONN, jest rozwiązaniem hybrydowym korzystającym z kompilatora ABY. Poprawia on wydajność protokołu MiniONN, zastępując złożoną fazę generowania trójek Bravera przez częściowo zaufaną trzecią stronę (*semi-honest third party*, STP), która odpowiedzialna jest za dostarczanie źródła niezależnej losowości dla obu stron.

Gazelle [22] (USENIX 2018) wprowadza kolejne optymalizacje konwersji przejść pomiędzy kryptosystemami oraz zoptymalizowane operacje SIMD (*single instruction multiple data*) dla dodawania oraz mnożenia macierzy, osiągając w ten sposób 20–30-krotnie większą wydajność w fazie online w porównaniu z poprzednimi rozwiązaniami (MiniONN oraz Chameleon).

XONN [30] (USENIX 2019) jest systemem stworzonym przez Microsoft, korzystającym z koncepcji binarnych sieci neuronowych (*binary neural network*, BNN), w których wagi i aktywację przyjmują tylko dwie wartości, ± 1. Dodatkowo protokół XONN kładzie nacisk na syntezę układu przy użyciu jak największej liczby bramek XNOR, które pozwalają na wykonywanie mnożenia bez

konieczności wykonania nieświadomego transferu (OT). W porównaniu z poprzednimi rozwiązaniami, gdzie liczba rund nieświadomych transferów była liniowa co do liczby warstw w NN, w XONN liczba rund jest stała, niezależnie od liczby warstw NN. Wynikiem tego jest siedmiokrotne przyspieszenie w porównaniu z Gazelle i około stukrotne przyspieszenie w porównaniu z MiniONN. XONN pozwala na interpretację wysokopoziomowych modeli NN bezpośrednio z biblioteki Keras.

CrypTFlow [24] (2019) to kolejny system stworzony przez Microsoft, składającym się z trzech komponentów. Pierwszym (Athos) jest kompilator pozwalający na zamianę dowolnego modelu NN zareprezentowanego przy użyciu Tensorflow do nieświadomego protokołu MPC. Drugim komponentem (Porthos) jest ulepszony protokół MPC pozwalający na nieświadome predykcje przy użyciu dużych sieci, jak ResNet50 lub DenseNet121, w czasie około 30 sekund. Trzecim komponentem (Aramis) jest protokół bazujący na założeniach bezpieczeństwa sprzętowego (np. Intel SGX), który zwiększa bezpieczeństwo oraz poprawia wydajność w porównaniu z rozwiązaniami czysto kryptograficznymi.

Delphi [26] (USENIX 2020) jest kolejną iteracją usprawnień w protokołach nieświadomej predykcji. W porównaniu z Gazelle Delphi zmniejsza ilość transferu danych z 560 MB do 60 MB oraz skraca czas predykcji z 82 sekund do 3,8 sekundy dla modelu ResNet-32 [20]. Jest to możliwe dzięki dwóm transformacjom: przeniesieniu większości obliczeń do fazy offline oraz zamianie nieliniowych funkcji aktywacji na bardziej MPC-przyjazne funkcje wielomianowe (kosztem około 1–5% spadku predykcji).

Przegląd rozwiązań w obszarze nieświadomych predykcji NN dostępny jest w pracy [10]. Wydajnościowe porównanie wszystkich rozwiązań jest problematyczne z powodu różnic modeli w wykonanych testach wydajnościowych oraz braku kodów źródłowych, co uniemożliwia przeprowadzenie reprodukowalności wyników.

7.6. Wnioski końcowe

Technologia umożliwiająca nieświadome obliczenia (MPC) znajduje zastosowanie w wielu obszarach, takich jak: biometryczne dopasowanie, rozpoznawanie twarzy, klasyfikacja danych, elektroniczne aukcje, internetowe wybory, zdalne diagnozy i bezpieczne wyszukiwanie.

Pomimo różnorodności protokołów używanych do unieświadamiania sieci neuronowych protokół *garbled circuit* (GC) używany jest w większości rozwiązań i nic nie wskazuje na to, aby sytuacja miała się zmienić. Dlatego w tym rozdziale skupiliśmy się właśnie na nim.

Najnowsze rozwiązania nieświadomych sieci neuronowych są rozwiązaniami hybrydowymi, tj. korzystają z wielu kryptosystemów w celu optymalizacji wykonania poszczególnych podprocedur.

W celu ułatwienia wytwarzania protokołów realizujących nieświadome obliczenia powstaje szereg kompilatorów pozwalających na wysokopoziomowy opis układu, automatyczny dobór kryptosystemów oraz konwersje pomiędzy nimi [10, 11, 13, 19].

Można zaobserwować dwa nurty rozwoju technologii nieświadomych obliczeń. Jeden z nich stara się zwiększyć wydajność, zmniejszyć obciążenie obliczeniowe, skrócić czas wykonania, przenieść jak największą ilość obliczeń do fazy offline oraz zmniejszyć narzut komunikacji. Drugi nurt stara się zwiększyć przystępność technologii, zapewnia kompilatory dla wysokopoziomowych języków oraz API do bezpośredniej integracji z bibliotekami i standardami, jak ONNX, TensorFlow, Keras i PyTorch.

O optymistycznej przyszłości protokołów nieświadomych obliczeń świadczy fakt, że są one rozwijane oraz wykorzystywane przez duże korporacje, jak Microsoft [16, 24] oraz Apple [1]. Dodatkowo najnowsze rozwiązania w obszarze *blockchain* bazują na technologii MPC w zakresie prywatnego wykonywania transakcji [6], a nawet inteligentnych kontraktów (*smart contracts*) [7].

Bibliografia

[1] *CSAM Detection - Technical Summary.* `https://www.apple.com/child-safety/pdf/CSAM_Detection_Technical_Summary.pdf`. (Accessed on 07/08/2022).

[2] Beaver D. *Efficient multiparty protocols using circuit randomization.* In *Annual International Cryptology Conference*, pp. 420–432. Springer, 1991.

[3] Beaver D., Micali S., Rogaway P. *The round complexity of secure protocols.* In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pp. 503–513. 1990.

[4] Ben-Or M., Goldwasser S., Wigderson A. *Completeness theorems for non-cryptographic fault-tolerant distributed computation.* In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pp. 351–371. 2019.

[5] Bost R., et al. *Machine learning classification over encrypted data. Cryptology ePrint Archive*, 2014.

[6] Bowe S., Gabizon A., Green M.D. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK.* In *International Conference on Financial Cryptography and Data Security*, pp. 64–77. Springer, 2018.

[7] Bowe S., et al. *Zexe: Enabling decentralized private computation.* In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 947–964. IEEE, 2020.

[8] Brakerski Z., Gentry C., Vaikuntanathan V. *(Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[9] Brakerski Z., Vaikuntanathan V. *Efficient fully homomorphic encryption from (standard) LWE. SIAM Journal on computing*, 43(2):831–871, 2014.

[10] Cabrero-Holgueras J., Pastrana S. *SoK: Privacy-preserving computation techniques for deep learning. Proceedings on Privacy Enhancing Technologies*, 2021(4):139–162, 2021.

[11] Chandran N., et al. *EzPC: programmable and efficient secure two-party computation for machine learning.* In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 496–511. IEEE, 2019.

[12] Chellapilla K., Puri S., Simard P. *High performance convolutional neural networks for document processing.* In *Tenth international workshop on frontiers in handwriting recognition.* Suvisoft, 2006.

[13] Demmler D., Schneider T., Zohner M. *ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.*

[14] Evans D., et al. *A pragmatic introduction to secure multi-party computation. Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018.

[15] Giaro K. *Elementy kwantowego modelu obliczeń i algorytmiki kwantowej: łagodne wprowadzenie do informatyki kwantowej.* Wydawnictwo Naukowe OWSIiZ.

[16] Gilad-Bachrach R., et al. *Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy.* In *International conference on machine learning*, pp. 201–210. PMLR, 2016.

[17] Goldreich O., Micali S., Wigderson A. *How to play any mental game, or a completeness theorem for protocols with honest majority.* In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pp. 307–328. 2019.

[18] Gupta S., et al. *SPARK: a parallelizing approach to the high-level synthesis of digital circuits.* Springer Science & Business Media, 2007.

[19] Hastings M., et al. *Sok: General purpose compilers for secure multi-party computation.* In *2019 IEEE symposium on security and privacy (SP)*, pp. 1220–1237. IEEE, 2019.

[20] He K., et al. *Deep residual learning for image recognition.* In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778. 2016.

[21] Holzer A., et al. *Secure two-party computations in ANSI C*. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 772–783. 2012.

[22] Juvekar C., Vaikuntanathan V., Chandrakasan A. {*GAZELLE*}*: A low latency framework for secure neural network inference*. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1651–1669. 2018.

[23] Keller M. *MP-SPDZ: A Versatile Framework for Multi-Party Computation*. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. URL `https://doi.org/10.1145/3372297.3417872`.

[24] Kumar N., et al. *Cryptflow: Secure tensorflow inference*. In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 336–353. IEEE, 2020.

[25] Liu J., et al. *Oblivious neural network predictions via minionn transformations*. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pp. 619–631. 2017.

[26] Mishra P., et al. *Delphi: A cryptographic inference service for neural networks*. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2505–2522. 2020.

[27] Paillier P. *Public-key cryptosystems based on composite degree residuosity classes*. In *International conference on the theory and applications of cryptographic techniques*, pp. 223–238. Springer, 1999.

[28] Rabin M.O. *How to exchange secrets with oblivious transfer*. *Cryptology ePrint Archive*, 2005.

[29] Riazi M.S., et al. *Chameleon: A hybrid secure computation framework for machine learning applications*. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pp. 707–721. 2018.

[30] Riazi M.S., et al. {*XONN*}*:*{*XNOR-based*} *Oblivious Deep Neural Network Inference*. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1501–1518. 2019.

[31] Rouhani B.D., Riazi M.S., Koushanfar F. *Deepsecure: Scalable provably-secure deep learning*. In *Proceedings of the 55th annual design automation conference*, pp. 1–6. 2018.

[32] Schoenmakers B. *MPyC–Python Package for Secure Multiparty Computation*. In *Workshop on the Theory and Practice of MPC*. 2018.

[33] Songhori E.M., et al. *Tinygarble: Highly compressed and scalable sequential garbled circuits*. In *2015 IEEE Symposium on Security and Privacy*, pp. 411–428. IEEE, 2015.

[34] Wang X., Malozemoff A.J., Katz J. *EMP-toolkit: Efficient MultiParty computation toolkit*. `https://github.com/emp-toolkit`, 2016.

[35] Wu D.J., et al. *Privately evaluating decision trees and random forests. Cryptology ePrint Archive*, 2015.

[36] Yao A.C. *Protocols for secure computations*. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pp. 160–164. IEEE, 1982.