Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Intytut Automatyki i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Inteligentne Systemy

Rozproszone uczenie maszynowe w sieci mikrokontrolerów

Grzegorz Kuduk
Numer albumu 277320

promotor
dr inż. Mariusz Kamola

WARSZAWA 2022

Rozproszone uczenie maszynowe w sieci mikrokontrolerów

Streszczenie. Celem pracy było zbadanie możliwości sieci mikrokontrolerów w sferze rozproszonego uczenia maszynowego przeprowadzanego na urządzeniach komunikujących się ze sobą bezpośrednio, dążącego do zbieżnego modelu całej sieci na każdym z nich.

W ramach badań przeprowadzono symulacje w środowisku Python z wykorzystaniem bibliotek TensorFlow i Keras. Podczas nich przeanalizowano dwie metody rozproszonego uczenia maszynowego. Pierwszą z nich było scalanie wag pomiędzy modelami w sposób zarówno synchroniczny, czyli analogiczny do rozwiązań korzystających z serwera odpowiedzialnego za agregację i uśrednianie modeli oraz rozsyłanie globalnego modelu wynikowego, jak i w sposób imitujący bezpośrednią komunikację asynchroniczną pomiędzy urządzeniami bez pośredniczącego serwera. Drugą metodą było popularne uśrednianie gradientów przeprowadzone w podobny sposób asynchroniczny. Symulacje przeprowadzono na dwóch zbiorach danych - zbiorze klasyfikującym ceny domów i zbiorze ręcznie pisanych cyfr MNIST. Porównano precyzje optymalizatorów Stochastic Gradient Descent i Adam.

Zaimplementowano także rzeczywistą sieć mikrokontrolerów złożoną z urządzeń Arduino Nano 33 BLE Sense komunikujących się pomiędzy sobą bezprzewodowo za pomocą Bluetooth Low Energy i przebadano ją pod kątem możliwości realizacji takiego rozwiązania w sieci urządzeń niezależnych od serwera. Badania implementacji wykazały, że urządzenia radzą sobie z uzyskaniem modeli o zadowalających precyzjach predykcji pomimo problemów z odczytywaniem danych przez protokół BLE.

Słowa kluczowe: Uczenie Maszynowe, Mikrokontroler, Edge Computing, TinyML

Distributed machine learning in a microcontroller network

Abstract. The goal of this work was to study the possibilities of a microcontroller network in the field of distributed machine learning carried out on devices communicating each other directly, aiming to converge models of the entire network on each one of them.

As part of the research, simulations were conducted in the Python environment using the TensorFlow and Keras libraries. During them two distributed machine learning methods were analyzed. The first was merging the weights between models in both a synchronized way, that is just like in solutions using a server responsible for aggregating models, averaging them and then distributing the global resulting model, as well as in a way that mimics direct asynchronous communication between devices without an intermediary server. The other one was the popular gradient averaging carried out in a similar asynchronous manner. Simulations were carried out on two datasets - a set classifying prices of houses and a set of handwritten digits, MNIST. The optimizers Stochastic Gradient Descent and Adam were compared in terms of accuracy.

A real-world microcontroller network consisting of Arduino Nano 33 BLE Sense devices communicating wirelessly with each other via Bluetooth Low Energy was also implemented and tested for the feasibility of such a solution in a network of server-independent devices. Tests of the implementation showed that the devices coped with obtaining models with satisfactory accuracy despite problems with reading data through the BLE protocol.

Keywords: Machine Learning, Federated Learning, Microcontrollers, Edge Computing, TinyML

Spis treści

1. Wstęp

Stale rozwijająca się dziedzina uczenia maszynowego znajduje swoje miejsce w coraz mniejszych urządzeniach. Obecnie pojawia się coraz więcej aplikacji uczenia maszynowego nawet na mikrokontrolerach - urządzeniach kojarzonych głównie z pobieraniem danych z sensorów i innymi zadaniami niewymagającymi intensywnych obliczeń.

Na popularności zyskuje również paradygmat rozproszonego uczenia maszynowego zwany sfederowanym uczeniem [1], w którym algorytmy są uruchamiane na wielu węzłach w sieci. Dzieje się tak ze względu na jego zalety skalowalności i redundancji. Ponadto, zdecentralizowane podejście pozwala na przechwytywanie danych treningowych z wielu fizycznych urządzeń z czujnikami, co zapewnia lepszą reprezentację rzeczywistych danych napotykanych przez system.

Niniejsza praca o tytule “Rozproszone uczenie maszynowe w sieci mikrokontrolerów” ma na celu zbadanie możliwości sieci mikrokontrolerów w sferze rozproszonego uczenia maszynowego przeprowadzanego na urządzeniach komunikujących się ze sobą bezpośrednio, dążącego do zbieżnego modelu całej sieci na każdym z nich. Tematyka pracy ściśle wiąże się z pojęciami przetwarzania brzegowego (ang. edge computing) oraz TinyML.

TinyML (tiny machine learning) [2] to szerokie pojęcie dziedziny technologii i aplikacji uczenia maszynowego na urządzeniach małego formatu - mikrokontrolerach - stosowanych najczęściej w celu przeprowadzania analizy danych z sensorów blisko ich źródła. Jednak zdecydowana większość dotychczasowych rozwiązań w tej sferze opiera się na przeprowadzaniu wyłącznie wnioskowania na gotowym modelu na mikrokontrolerze, podczas gdy wykorzystany model jest uczony w całości na innym urządzeniu bardziej do tego przystosowanym, czyli na komputerze lub w chmurze. Do popularnego rozwiązania do tego służącemu należy narzędzie TensorFlow Lite, które taki model kompresuje do małego rozmiaru z wykorzystaniem technik obcinania połączeń między neuronami i kwantyzacji wag.

Niektóre z innych rozwiązań [3], [4] przeprowadzają trening na mikrokontrolerach i uzyskują efekt rozproszonego uczenia maszynowego poprzez wykorzystanie centralnego serwera, z którym mikrokontrolery są w stałym kontakcie. Serwer ten jest odpowiedzialny za uzyskiwanie wspólnego, globalnego modelu wykorzystując modele lokalne wszystkich urządzeń i rozesłanie go do całej sieci. Wynikiem jest sieć urządzeń posiadających jednakowy model, co jest pożądane, ale możliwe tylko poprzez synchronizację procesów uczenia i wymiany modeli oraz stałemu kontaktowi urządzeń z serwerem. Niestety oznacza to także, że w przypadku awarii takiego serwera cała sieć zostaje pozbawiona możliwości wspólnego uczenia.

Z tego względu zaimplementowane w ramach tej pracy rozwiązanie opiera się o sieć mikrokontrolerów, w której każde urządzenie jest niezależne od serwerów i wciąż jest w stanie dążyć z modelem lokalnym do zbieżnego modelu globalnego całej sieci. Ta-

kiego typu rozwiązanie zapewnia również prywatność danych, ponieważ dane urządzenie dzieli się jedynie danymi określającymi model (wagi lub gradienty), przechowując dane wykorzystane do jego treningu lokalnie [5].

W pierwszej kolejności przeprowadzono symulacje w środowisku Python z wykorzystaniem bibliotek TensorFlow i Keras w celu przebadania algorytmów umożliwiających osiągnięcie takiego rezultatu. Następnie analogicznie zastosowano te algorytmy w rzeczywistej sieci mikrokontrolerów złożonej z trzech urządzeń Arduino Nano 33 BLE Sense. W celu uzyskania powtarzalnych wyników, które nie są zależne od otoczenia, dane treningowe dostarczano do urządzeń za pomocą serwera Python połączonego z nimi przez port szeregowy. Do badań wykorzystano dwa zbiory danych służące do klasyfikacji próbek: do 2 klas dla zbioru cen domów oraz do 10 klas dla zbioru cyfr pisanych ręcznie MNIST.

2. Zbiór wiedzy

Zdecydowana większość dotychczasowych prac w kategorii rozproszonego uczenia maszynowego na urządzeniach małego formatu, czyli TinyML, opierała się na wykorzystaniu modeli wytrenowanych w całości w środowisku do tego przystosowanym, takim jak komputer z dedykowaną kartą graficzną lub chmura. Adaptacja takiego modelu do użytku na mikrokontrolerze polega na obcięciu mało znaczących wag (ang. pruning) oraz ich kwantyzacji (ang. quantization) w celu redukcji rozmiaru [6]. Biblioteka TensorFlow Lite [7] należy do popularnych narzędzi oferujących takie możliwości.

Wykorzystanie takiego rozwiązania pozwala na wdrożenie modeli w skompresowanej formie bez większego spadku ich dokładności, ale wymaga wcześniejszego nauczenia poza docelowym urządzeniem oraz nie pozwala na dalszy trening tych modeli w celu zwiększenia precyzji predykcji lub adaptacji do potencjalnych zmian środowiska, w którym urządzenie się znajduje, a więc i danych, które otrzymuje.

Praca napisana przez Disabato i Roveri [8] przedstawiła uczenie inkrementalne przeprowadzone na mikrokontrolerach algorytmem opierającym się na uczeniu przez transfer (ang. transfer learning) i K najbliższych sąsiadów (ang. k-nearest neighbor). Badania przeprowadzono dla zbiorów obrazów i audio, ale użyto w nich wytrenowany wcześniej ekstraktor cech. Wykorzystanymi urządzeniami były Raspberry Pi 3B+ i STM32F7 o 512 kB pamięci RAM i 2 MB pamięci flash - pojemności aż 2 razy większe od wykorzystanych w tej pracy mikrokontrolerów, ale wciąż porównywalne do nich.

Praca Giméneza i innych [3] jest stosunkowo bliska zaimplementowanemu w ramach pracy dyplomowej rozwiązaniu. Autorzy zbadali program rozproszonego uczenia maszynowego mający za zadanie wykrycie kluczowych słów. Całość procesu, czyli nagrywanie próbek, uczenie oraz wnioskowanie przeprowadzili na mikrokontrolerach. Badania wykonali na trzech urządzeniach Arduino Nano 33 BLE Sense, czyli takich samych jak niniejsza praca, ale architektura ich rozwiązania skupiła się na wykorzystaniu serwera odpowiedzialnego za agregację, scalanie i rozsyłanie modeli - urządzenia nie komunikowały się pomiędzy sobą bezpośrednio. Takie rozwiązanie sprawdza się dość dobrze, ale wymaga od każdego urządzenia w sieci stałego kontaktu z serwerem, przez co w realnym zastosowaniu może ograniczać rozmieszczenie tych urządzeń i wprowadza element krytyczny dla funkcjonowania całości.

Niektóre prace w dziedzinie rozproszonego uczenia maszynowego zbadały różne metody uzyskania wynikowego modelu z wielu modeli, jak uczenie o wyrównanych cechach [9] czy uśrednianie maskowanych gradientów [10]. Zbadane w nich algorytmy uzyskały wyniki lepsze od standardowych, ale wykonano je na bardzo dużych sieciach z dużą ilością danych treningowych. Dodatkowo podobnie jak wspomniana powyżej praca [3] również skupiły się na strukturze sieci opartej o centralny serwer.

3. Technologia

W celu implementacji rozproszonego uczenia maszynowego w rzeczywistej sieci mikrokontrolerów skorzystano z urządzeń Arduino, które umożliwiły zbudowanie komunikacji pomiędzy sobą na Bluetooth Low Energy. Rozdział ten opisuje i przybliża te technologie.

3.1. Arduino

Arduino jest znane ze swojej oferty szerokiej gamy powszechnie stosowanych urządzeń małego formatu - mikrokontrolerów.

Urządzenie wykorzystane do implementacji sieci mikrokontrolerów to Arduino Nano 33 BLE Sense. Zostało ono wybrane ze względu na posiadane parametry:

- Procesor ARM Cortex-M4 32-bit 64 Mhz - jest to jeden z szybszych, dostępnych w czasie pisania tej pracy, procesorów w urządzeniach tego formatu. Umożliwia on wykonywanie operacji na nawet dużych sieciach neuronowych z dużą prędkością.
- 256 kB pamięci RAM - pojemność pozwala przechowywać nawet bardzo duże sieci neuronowe na urządzeniu z dużym zapasem na przykład na “sztucznie” pozyskane zestawy danych wykorzystywane do procesu uczenia maszynowego.
- Bluetooth 5.0 Low Energy - urządzenie zawiera moduł do komunikacji BLE zainstalowany “out of the box”, dzięki czemu pozwala na prostą komunikację z innymi urządzeniami z wykorzystaniem powszechnie stosowanego protokołu komunikacji bezprzewodowej.

Nano 33 BLE Sense posiada 1 MB pamięci flash, co udostępnia jeszcze więcej pamięci do użytku, która jednak nie jest wykorzystywana bezpośrednio przez zaimplementowany program, ponieważ 256 kB szybszej pamięci RAM w pełni wystarczyło na jego poprawne funkcjonowanie.

Dodatkowo mikrokontroler posiada wsparcie TensorFlow Lite, co oznacza, że w bardzo łatwy sposób może on obsłużyć modele zbudowane w TensorFlow, które przeszły odpowiednią konwersję. Modele takie cechują się znacznie zmniejszoną wielkością zajmowanej pamięci - rzędu kilku kilobajtów. Niestety o ile jest to wygodnym i wydajnym sposobem na przeprowadzanie wnioskowania na gotowych modelach, TensorFlow Lite nie umożliwia tworzenia i trenowania od podstaw lub nawet dotrenowania istniejących sieci, co wyklucza zastosowanie tego sposobu implementacji uczenia maszynowego w tej pracy.

Jak wskazuje ostatni człon nazwy Arduino Nano 33 BLE Sense, urządzenie posiada również szereg sensorów - akcelerometr, żyroskop, magnetometr, barometr i inne - pozwalających na zbieranie rozmaitych danych z otoczenia. Dane te mogą zostać wykorzystane do lokalnego trenowania różnych modeli uczenia maszynowego, ale w przypadku zaimplementowanego rozwiązania zdecydowano się na wykorzystanie gotowych zestawów danych, ponieważ zbieranie odpowiedniej ilości danych z otoczenia co prawda pomogłoby odtworzyć warunki przy praktycznym przykładzie zastosowania, ale każda próba wymaga-

łaby zwiększonego nakładu czasowego oraz nie zapewniałaby dokładnej powtarzalności danych.

3.2. Bluetooth Low Energy

Komunikacja bezprzewodowa pomiędzy wykorzystanymi urządzeniami Arduino odbywa się poprzez Bluetooth Low Energy.

Bluetooth Low Energy (BLE) to technologia wywodząca się, z tradycyjnego Bluetooth (Bluetooth Classic). Pomimo podobnej nazwy obu technologii, nie są one ze sobą kompatybilne, ale obie mogą być wykorzystywane przez dane urządzenie. Tak jak nazwa BLE wskazuje, cechuje się ona znacznie mniejszym zużyciem energii, dzięki czemu jest powszechnie wykorzystywana w urządzeniach z bateriami o niewielkiej pojemności lub ograniczonym poborem prądu, takich jak mikrokontrolery.

Niskie zużycie energii uzyskiwane jest między innymi poprzez usypianie modułu BLE, gdy nie jest on w użytku i włączanie go tylko wtedy, kiedy jest potrzebny. Ponadto, w porównaniu do klasycznego Bluetooth, który służy do ciągłej, dwustronnej komunikacji, BLE transferuje mniejsze pakiety danych w krótkich okresach czasowych.

Oprócz tego wersja Low Energy wyróżnia się swoją architekturą, w której urządzenia dzielą się na role:

- Central - urządzenie może wykonać skan w celu znalezienia urządzeń Peripheral, łączyć się z nimi i odczytywać ich dane lub zapisywać swoje dane na nich
- Peripheral - urządzenie może wystawiać lub rozgłaszać dane i odebrać połączenia od innych urządzeń.

Mimo wydzielenia ról, dane urządzenie może działać w obu rolach naraz. Fakt ten został wykorzystany w zaimplementowanym rozwiązaniu z wykorzystaniem mikrokontrolerów Arduino, aby każdemu z urządzeń umożliwić zarówno pobieranie danych z sieci mikrokontrolerów jak i udostępnianie do niej swoich.

Aby zrozumieć jak urządzenia komunikują się ze sobą poprzez BLE, należy najpierw zapoznać się z następującą terminologią:

- Generic Attribute Profile (GATT) - struktura danych urządzenia Peripheral udostępniana połączonym urządzeniom Central.
- Charakterystyka - identyfikowana przez UUID. Zawiera wartość transferowaną pomiędzy urządzeniami Peripheral i Central oraz dotyczące niej własności i deskryptory (metadane). Wartość reprezentowana jest jako ciąg bajtów.
- Serwis - kolekcja charakterystyk. Identyfikowany przez UUID.
- Universally Unique Identifier (UUID) - 128-bitowy identyfikator w określonym standardzie.

Komunikacja pomiędzy urządzeniem Central a Peripheral wymaga odpowiednich ustawień początkowych. Najpierw należy wybrać identyfikator UUID dla wykorzystywanych serwisów i charakterystyk oraz odpowiednio je z nim zarejestrować w module

BLE - do przekazywania danych należy stworzyć charakterystykę, która “podpina się” pod serwis. Następnie można ustawić wybrany pojedynczy serwis jako reklamowany i rozpocząć jego rozgłaszanie. Urządzenia w zasięgu, które skanują w poszukiwaniu innych urządzeń, widzą nazwę, adres MAC oraz UUID reklamowanego serwisu, po których mogą się połączyć. Po udanej próbie połączenia mogą odczytać atrybuty urządzenia, czyli między innymi dostępne charakterystyki, i zacząć odczytywać dane.

3.3. Uczenie maszynowe na Arduino

Koncepcja uczenia maszynowego na mikrokontrolerach nie jest nowa, lecz wykonywanie treningu na nich wciąż nie jest zbyt popularne. Zdecydowana większość istniejących zastosowań opiera się na wykonywaniu jedynie procesu inferencji na mikrokontrolerze przy wykorzystaniu modeli, które zostały wcześniej nauczone w standardowym, lepiej do tego przygotowanym środowisku, takim jak program Python korzystający z bibliotek TensorFlow i Keras.

Chcąc stworzyć nową sieć neuronową oraz przeprowadzić na niej trening wyłącznie na urządzeniu Arduino, jest się zmuszonym do przeszukiwania mało popularnych bibliotek z, w większości przypadków, wybrakowanymi funkcjonalnościami lub nawet do tworzenia własnej biblioteki od zera. Mimo to istnieje kilka nadających się do użytku w rozważanym aspekcie.

Do implementacji uczenia maszynowego na Arduino wybrano bibliotekę NeuralNetworks [11]. Oferuje ona dużo możliwości w porównaniu do innych bibliotek tego typu na Arduino. Między innymi umożliwia konstruowanie sieci neuronowych o wielu warstwach z przypisaniem wybranej funkcji aktywacji do każdej z nich, co jest oczywiste w standardowych bibliotekach jak TensorFlow, ale nie zawsze przewidziane w ich odpowiednikach na Arduino.

Biblioteka ta opiera się na podstawowym algorytmie propagacji w przód i propagacji wstecz. Dodatkowo biblioteka ta pozwala na wybranie opcji przechowywania wag jako liczb o zmniejszonej precyzji - 2-bajtowych float16 - dzięki czemu w przypadku większych modeli i mniej pojemnych urządzeń istnieje możliwość oszczędności w zakresie zajmowanej przez model pamięci. Opcja ta nie została jednak wykorzystana w tej pracy, ponieważ wykorzystane urządzenie zapewniło wystarczająco dużo pamięci RAM.

4. Implementacja

Do symulacji w środowisku Python w wersji 3.10.4 wykorzystano biblioteki TensorFlow [12] 2.8.0 i Keras [13] 2.8.0 do uczenia maszynowego oraz biblioteki NumPy 1.22.3, MatPlotLib 3.5.3, Pandas 1.4.2, SciKit-Learn 1.1.1 do różnych operacji związanych z odczytywaniem plików csv, przetwarzaniem danych, przedstawianiem wyników i tym podobnych. Operacje uczenia maszynowego w symulacjach wykonano za pomocą środowiska CUDA 11.2 na karcie NVIDIA GeForce RTX 2070 Super 8GB.

Program do uczenia rozproszonego uczenia maszynowego na urządzeniach Arduino Nano 33 BLE Sense napisano wykorzystując bibliotekę NeuralNetworks [11] oraz ArduinoBLE [14] w wersji 1.3.1. Komunikację pomiędzy serwerem Python i urządzeniami Arduino zaimplementowano poprzez PySerial w wersji 3.5. Do kompilacji oraz nagrywania na docelowe urządzenia skorzystano z wtyczki do Visual Studio Code - PlatformIO [15].

Symulacje, serwer Python łączący się z urządzeniami Arduino oraz program na nie wgrywany stworzono i uruchamiano w środowisku Visual Studio Code 1.71.

4.1. Zbiory danych

W celu zbadania rozproszonego uczenia maszynowego wybrano dwa zbiory danych odpowiednie dla zadania klasyfikacji. Pierwszym z nich jest zbiór z 10 atrybutami opisującymi domy wystawione na sprzedaż w Stanach Zjednoczonych wraz z określeniem czy dany dom ma cenę poniżej czy powyżej mediany wszystkich cen. Oryginalny zbiór firmy Zillow zawiera bardzo dużą liczbę atrybutów oraz bezpośrednio określa ceny nieruchomości. W eksperymentach wykorzystano dane przetworzone [16] na problem klasyfikacji z odniesieniem do mediany wszystkich cen, określone zredukowaną liczbą atrybutów. Do tych atrybutów należą:

- LotArea - powierzchnia działki
- OverallQual - ogólna ocena jakości
- OverallCond - ogólna ocena stanu
- TotalBsmtSF - całkowita powierzchnia piwnicy
- FullBath - liczba “pełnych” łazienek
- HalfBath - liczba “częściowych” łazienek
- BedroomAbvGr - liczba sypialni nad ziemią
- TotRmsAbvGrd - całkowita liczba pokoi nad ziemią
- Fireplaces - liczba kominków
- GarageArea - powierzchnia garażu

Zestaw składa się z 1461 rekordów. Dane zostały przeskalowane do zakresu o wartościach 0 do 1, gdzie 0 to najmniejsza wartość w kategorii, a 1 największa.

Drugim wykorzystanym zbiorem danych jest zbiór cyfr MNIST [17]. Zbiór ten składa się z obrazków 28 na 28 pikseli ręcznie napisanych cyfr, których przykłady zaprezentowano na

rysunku 4.1. Zestaw danych zawiera 60000 rekordów treningowych oraz 10000 rekordów testowych.

Rysunek 4.1. Przykłady obrazów cyfr ze zbioru danych do rozpoznawania cyfr MNIST. Źródło: `https://en.wikipedia.org/wiki/MNIST_database`

Każdy z rekordów w zbiorze zawiera odpowiadającą obrazkowi tablicę dwuwymiarową o rozmiarze 28 na 28, gdzie każdy element to wartość całkowita od 0 do 255, reprezentująca poziom szarości danego piksela. W celu wykorzystania do uczenia maszynowego, każdą z tych wartości znormalizowano do liczby zmiennoprzecinkowej z przedziału od 0 do 1, gdzie 0 odpowiada początkowej wartości 0, a 1 odpowiada początkowej wartości 255 i następnie tablicę tych wartości "spłaszczono" do jednowymiarowej tablicy o rozmiarze 784 (28x28). Przetworzone w taki sposób liczby wykorzystano jako dane wejściowe do sieci neuronowej. Rekordy zawierają również oznaczenie w postaci cyfry od 0 do 9 określające faktyczną cyfrę przedstawioną na obrazku. Oznaczenie to przekształcono do postaci tablicy 10 prawdopodobieństw o wartościach od 0 do 1, z których każde odpowiada kolejnej cyfrze, aby odpowiadało ono wyjściu sieci neuronowej klasyfikującej przynależność danego wejścia do 10 kategorii jednocześnie.

4.2. Symulacje TensorFlow

Symulacje scenariusza interakcji mikrokontrolerów zaimplementowano w programie Python z wykorzystaniem bibliotek TensorFlow i Keras. Każda symulacja składa się z dwóch części - wczytania danych i inicjalizacji modeli oraz treningu i ewaluacji (w jednej lub wielu rundach). W pierwszej kolejności zdefiniowane zostają kluczowe parametry symulacji oraz wybrany zostaje docelowy optymalizator, co przedstawiono we fragmencie kodu 4.1 (na przykładzie parametrów z testu z tabeli 5.3). Oznaczają one kolejno: ilość próbek ze zbioru danych na jeden model, liczbę epok, ilość rund procesu uczenia oraz

wartość parametru *mini_batch*, z którego korzysta funkcja trenująca biblioteki Keras. Wybierany jest również optymalizator (linijka 7).

Kod 4.1. Inicjalizacja parametrów i optymalizatora symulacji.

```
1  import tensorflow as tf
2
3  BATCH_SIZE = 300
4  NUM_EPOCHS = 10
5  NUM_ROUNDS = 4
6  MINI_BATCH_SIZE = 10
7  optim = tf.keras.optimizers.SGD(learning_rate=0.3)
```

Następnie załadowany zostaje jeden z dwóch wybranych zbiorów danych. We fragmencie kodu 4.2 przedstawiono wczytanie danych o klasyfikacji cen domów z pliku csv. Uzyskane dane poddane są obróbce w postaci normalizacji według atrybutów, widocznej w linijce 11. Jeśli dana symulacja wymaga podziału na 3 zestawy treningowe oraz zestaw testowy, zbiór zostaje podzielony w zależności od wartości *BATCH_SIZE* na odpowiednie listy rekordów (linijki 13-15 i 18-20). Oddzielony zostaje również zestaw testowy (linijki 16 i 21).

Kod 4.2. Wczytanie danych o klasyfikacji cen domów.

```
1  import pandas as pd
2  from sklearn import preprocessing
3
4  dataframe = pd.read_csv('data/housepricedata.csv')
5  dataset = dataframe.values
6
7  X = dataset[:,0:10]
8  Y = dataset[:,10]
9
10 min_max_scaler = preprocessing.MinMaxScaler()
11 X = min_max_scaler.fit_transform(X)
12
13 X1 = X[:BATCH_SIZE]
14 X2 = X[BATCH_SIZE:2*BATCH_SIZE]
15 X3 = X[2*BATCH_SIZE:3*BATCH_SIZE]
16 Xtest = X[1000:1400]
17
18 Y1 = Y[:BATCH_SIZE]
19 Y2 = Y[BATCH_SIZE:2*BATCH_SIZE]
20 Y3 = Y[2*BATCH_SIZE:3*BATCH_SIZE]
21 Ytest = Y[1000:1400]
```

Alternatywnie wczytany może zostać zbiór danych o klasyfikacji cyfr MNIST. We fragmencie kodu 4.3 zaprezentowano operacje tego działania. Zestaw z gotowym podziałem na zbiór treningowy i testowy również należy poddać obróbce. Dwuwymiarowa tablica o wymiarach 28 na 28 wartości całkowitych od 0 do 255 zostaje znormalizowana do

przedziału liczb zmiennoprzecinkowych od 0 do 1 (linijki 5 i 7) i następnie spłaszczona do tablicy jednowymiarowej o długości 784 (linijki 6 i 8). Wartości Y określające liczby przedstawiane przez dane wejściowe X zostają przekształcone z pojedynczej wartości całkowitej od 0 do 9 do tablicy 10 wartości określających przynależność danego rekordu do każdej z 10 klas (linijki 10-13). Zbiór danych treningowych zostaje podzielony analogicznie do fragmentu kodu 4.2.

Kod 4.3. Wczytanie danych o klasyfikacji cyfr MNIST.

```
1  from keras.datasets import mnist
2
3  (X, Y), (Xtest, Ytest) = mnist.load_data()
4
5  Xtest = Xtest / 255.0                          # Normalize
6  Xtest = Xtest.reshape(Xtest.shape[0], -1)    # Flatten
7  X = X / 255.0                    # Normalize
8  X = X.reshape(X.shape[0], -1)    # Flatten
9
10 Ytest = tf.keras.utils.to_categorical( \
11                          Ytest, num_classes=10, dtype='float32')
12 Y = tf.keras.utils.to_categorical( \
13                          Y, num_classes=10, dtype='float32')
```

W kolejnym kroku inicjalizowane i kompilowane są modele przy użyciu biblioteki Keras. Przykład przedstawiony we fragmencie kodu 4.4 dotyczy sieci o układzie {10, 4, 4, 1} zastosowanej do zbioru danych cen domów z rysunku 5.1.

Kod 4.4. Inicjalizacja modeli biblioteki Keras.

```
1  from keras.layers import Dense
2
3  def init_model():
4      model = keras.Sequential()
5      model.add(keras.Input(shape=(10,)))
6      model.add(Dense(4, activation='relu', \
7                 use_bias=True, bias_initializer='ones'))
8      model.add(Dense(4, activation='relu', \
9                 use_bias=True, bias_initializer='ones'))
10     model.add(Dense(1, activation='sigmoid', \
11                use_bias=True, bias_initializer='ones'))
12     return model
13
14 model1 = init_model()
15 model1.compile(loss='binary_crossentropy', \
16                 optimizer=optim, metrics=['accuracy'])
```

Następną fazą jest faktyczna symulacja, czyli trening i ewaluacja modeli. Główną pętlę wykonywaną przez ilość rund określoną przez *NUM_ROUNDS* przedstawiono we fragmencie kodu 4.5. W każdej rundzie każdy z modeli zostaje poddany treningowi na kolejnym

fragmencie przydzielonego zestawu danych i obliczana zostaje precyzja. W dalszej części pętli wykonane zostają ewentualne operacje scalania wag lub uśredniania gradientów. W celu zapewnienia powtarzalności testów ustawiane są wartości *seed* bibliotek TensorFlow i NumPy.

Kod 4.5. Pętla treningu i ewaluacji modeli.

```
import numpy as np

tf.random.set_seed(1234)
np.random.seed(1234)

ITER_BATCH = np.int32(BATCH_SIZE / NUM_ROUNDS)
for i in range(0, NUM_ROUNDS):
    n = np.int32(ITER_BATCH * i)
    model1.fit(X1[n : n + ITER_BATCH], Y1[n : n + ITER_BATCH], \
               epochs=NUM_EPOCHS, batch_size=MINI_BATCH_SIZE))
    ...
    _, accuracy1 = model1.evaluate(Xtest, Ytest)
    ...
    # scalenie wag / usrednianie gradientow
```

We fragmencie kodu 4.6 zaprezentowano operacje wykonywane podczas scalenia wag trzech modeli w symulacji komunikacji synchronicznej. Wynikiem jest zestaw wag uśrednionych, które następnie nadpisują dotychczasowe wagi we wszystkich modelach.

Kod 4.6. Proces uśredniania wag w symulacji komunikacji synchronicznej.

```
weights1 = model1.get_weights()
...

weights_merged = []
for j in range(0, len(weights1)):
    layer_weights_merged = (weights1[j] + weights2[j] \
                            +  weights3[j])/3.0
    weights_merged.append(layer_weights_merged)

model1.set_weights(weights_merged)
...
```

Alternatywnym zaimplementowanym sposobem scalania wag, który zastosowano w celu trochę wierniejszej reprezentacji komunikacji asynchronicznej, jest wykorzystanie średniej ważonej z wagami złotego podziału. Powodem wybrania tego podziału był jego wynikowy rozkład uzyskany poprzez dwie operacje scalenia wag. Wzory na wynikowe wagi tych operacji dla modelu 1 przedstawiono poniżej - w_i oznacza wagi modelu i, w_{m1}

wagi po pierwszej operacji scalenia i w_{m2} wagi po drugiej operacji:

$$w_{m1} = 0.618\,w_1 + 0.382\,w_2$$
$$w_{m2} = 0.618\,w_{m1} + 0.382\,w_3$$

Z powyższych równań, w przypadku wykonania obu operacji, rozkład uzyskanych wag wygląda następująco:

$$w_{m2} = 0.382\,w_1 + 0.236\,w_2 + 0.382\,w_3$$

Oznacza to, że nowe wagi dla modelu 1 składają się z równych części jego dotychczasowych wag oraz wag modelu wykorzystanego w ostatniej operacji i w 23.6% z wag modelu użytego w poprzedniej operacji. dzięki temu uzyskano stosunkowo równy podział ważności wag scalanych modeli z większą ważnością wag oryginalnych oraz tych, które uzyskano niedawno.

Implementację tego sposobu przedstawiono we fragmencie kodu 4.7, w którym dla każdego z modeli wyliczany zostaje nowy zestaw wag z wykorzystaniem dotychczasowych wag modelu oraz wag innych modeli.

Kod 4.7. Proces uśredniania wag w symulacji komunikacji asynchronicznej.

```
1  weights1 = model1.get_weights()
2  ...
3
4  weights_merged1 = []
5  ...
6  for j in range(0, len(weights1)):
7      layer_weights_merged1 = 0.618*weights1[j] + 0.382*weights2[j]
8      layer_weights_merged1 = 0.618*layer_weights_merged1 \
9                              + 0.382*weights3[j]
10     weights_merged1.append(layer_weights_merged1)
11
12     layer_weights_merged2 = 0.618*weights2[j] + 0.382*weights3[j]
13     layer_weights_merged2 = 0.618*layer_weights_merged2 \
14                             + 0.382*weights1[j]
15     weights_merged2.append(layer_weights_merged2)
16
17     layer_weights_merged3 = 0.618*weights3[j] + 0.382*weights1[j]
18     layer_weights_merged3 = 0.618*layer_weights_merged3 \
19                             + 0.382*weights2[j]
20     weights_merged3.append(layer_weights_merged3)
21
22 model1.set_weights(weights_merged1)
23 ...
```

Drugą zaimplementowaną metodą rozproszonego uczenia maszynowego jest metoda

uśredniania gradientów, która również korzysta ze średniej ważonej ze złotym podziałem wag. Gradienty danego modelu zostają uśredniane z gradientami pozostałych modeli analogicznie do wag w symulacji komunikacji asynchronicznej. We fragmencie kodu 4.8 przedstawiono sposób uzyskania i aplikacji gradientów do modeli za pomocą funkcji *get_grads* oraz *apply_grads*, które w całości zaprezentowano w załączniku 4. Pierwsza z tych funkcji oblicza i zwraca gradienty na podstawie wskazanego modelu i jego poprzednich wag oraz zeruje gradienty poniżej parametru *GRAD_THRESHOLD*, o ile został on w danej symulacji zdefiniowany. Zerowanie gradientów odzwierciedla ich obcięcie, czyli pozbycie się mało istotnych zmian. Potencjalnie może to pozwolić na zmniejszenie ilości przekazywanych danych, co w środowisku o ograniczonej przepustowości kanału komunikacyjnego może być kluczowe. W tym celu również dołączona wersja funkcji *get_grads* zmniejsza precyzję wynikowych gradientów do 2-bajtowych liczb zmiennoprzecinkowych float16. Druga z przedstawionych funkcji - *apply_grads* - dodaje uzyskane gradienty do poprzednich wag modelu.

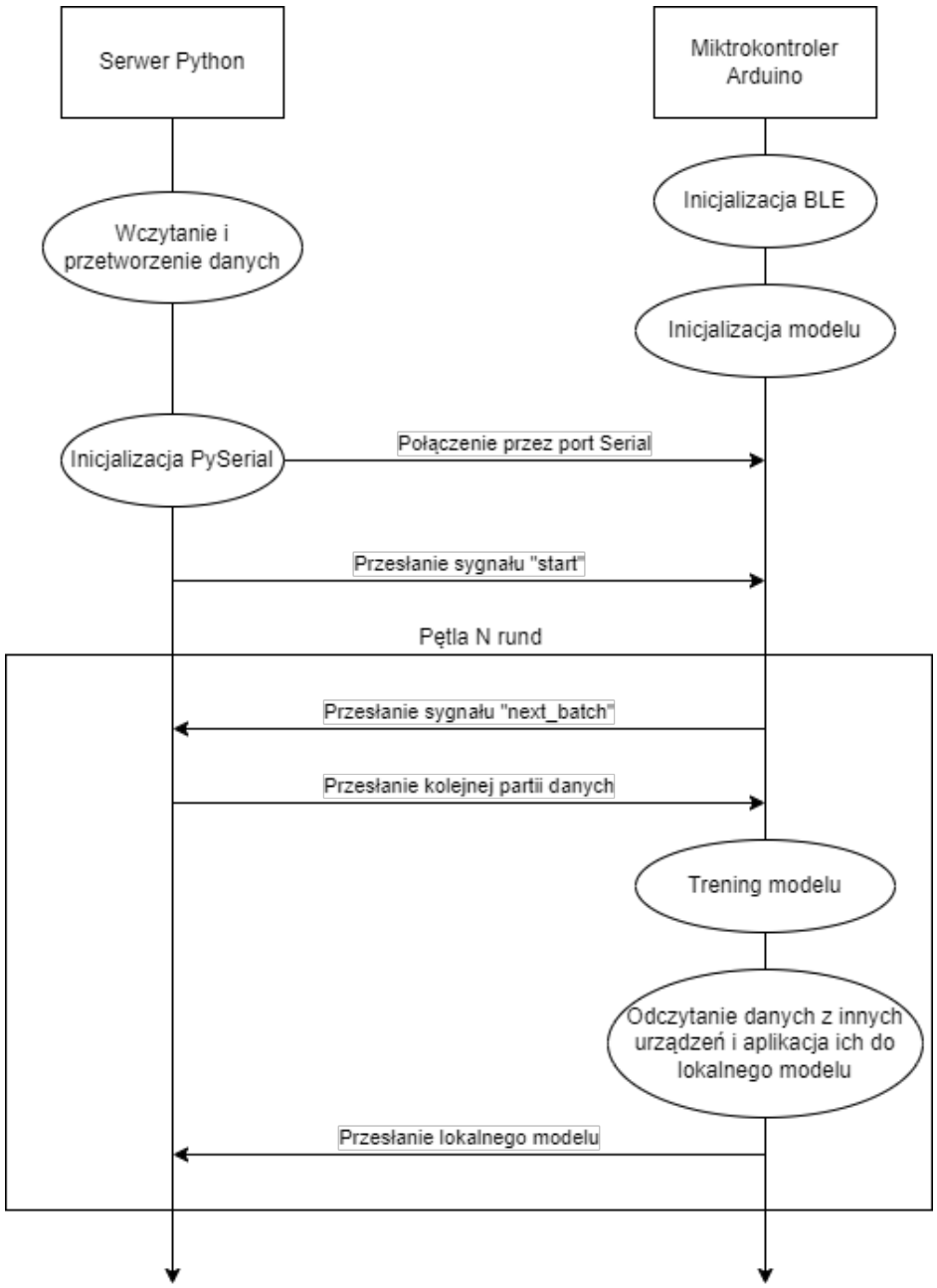
Kod 4.8. Proces aplikacji gradientów.

```
w1 = model1.get_weights()
...

for i in range(0, NUM_ROUNDS):
    ...

    weights1 = model1.get_weights()
    ...

    grads1, stats1 = get_grads(w1, weights1)
    grads2, stats2 = get_grads(w2, weights2)
    grads3, stats3 = get_grads(w3, weights3)

    for j in range(0, len(grads1)):
        layer_grads1 = 0.618*grads1[j] + 0.382*grads2[j]
        layer_grads1 = 0.618*layer_grads1 + 0.382*grads3[j]
        grads_merged1.append(layer_grads1)
        ...

    apply_grads(model1, w1, grads_merged1)
    ...

    w1 = model1.get_weights()
    ...
```

4.3. Serwer Python

Serwer Python jest odpowiedzialny za komunikację z mikrokontrolerami w celu zapisywania logów oraz dostarczania danych treningowych. Na rysunku 4.2 przedstawiono

diagram sekwencyjny komunikacji pomiędzy serwerem i urządzeniami Arduino poprzez porty Serial.



Rysunek 4.2. Diagram sekwencyjny komunikacji przez port Serial pomiędzy Serwerem Python i danym urządzeniem Arduino

W pierwszej kolejności zostają wczytane i przetworzone dane treningowe, analogicznie do implementacji odczytu danych w programach służących do symulacji, opisanych we wcześniejszej części tego rozdziału. Potem następuje sprawdzenie dostępnych przez porty serial urządzeń za pomocą funkcji *comports()* i inicjalizacja połączeń z nimi, przedstawione we fragmencie kodu 4.9. Połączenia zostają otwarte na odpowiedniej przepustowości, w tym przypadku o wartości standardowej 9600.

Kod 4.9. Inicjalizacja połączeń serwera z mikrokontrolerami.

```
import serial
from serial.tools.list_ports import comports

def openSerials(available_devices):
    serials = []
    for dev in available_devices:
        serials.append(serial.Serial(dev.device, baudrate=9600))
    return serials

devices = comports()
serials = openSerials(devices)
```

W momencie uruchomienia serwera mikrokontrolery są już gotowe (po inicjalizacji modułu Bluetooth Low Energy i modelu sieci neuronowej) i czekają na przesłanie od serwera bajtów odpowiadających ciągowi znaków "start". Pozwala to na synchronizację rozpoczęcia procesu rozproszonego uczenia maszynowego na urządzeniach Arduino z serwerem, co jest ważne zarówno ze względu na interakcje pomiędzy tymi urządzeniami oraz zapisywanie logów po stronie serwera.

Po wczytaniu i przetworzeniu danych i następnie inicjalizacji połączeń przez porty serial, bufory zaczynają zbierać dane odbierane przez porty i serwer jest gotowy do dalszej pracy, więc przesyła bajty "start" do urządzeń za pomocą funkcji *sendStartSignalToDevices* przedstawionej we fragmencie kodu 4.10.

Kod 4.10. Funkcja wysyłająca sygnał "start" do urządzeń.

```
def sendStartSignalToDevices(serials):
    print('Starting␣training')
    for ser in serials:
        ser.write(b'start')

sendStartSignalToDevices(serials)
```

Zaraz po wykonaniu tych czynności serwer wpada w docelową pętlę, przedstawioną we fragmencie kodu 4.11, w której po odczekaniu sekundy na częściowe wypełnienie buforów komunikacji serial, parsuje ich zawartości za pomocą funkcji *parseSerialOutput*, zaprezentowaną we fragmencie kodu 4.12.

Kod 4.11. Pętla serwera zbierająca dane z urządzeń.

```
while(True):
    sleep(1)
    parseSerialOutput(serials)
```

Kod 4.12. Funkcja parseSerialOutput.

```
def parseSerialOutput(serials):
    for i, ser in enumerate(serials):
```

```
3           s = str(ser.read_all(), 'utf-8')
4           updateLogs(s, ser.port)
5           # parse the output s and respond to notable strings
6           cmdStart = s.find('::')
7           while (cmdStart != -1):
8               cmdEnd = s[cmdStart:].find('\r')
9               cmd = s[cmdStart+2:cmdStart+cmdEnd]
10              print(f'Received cmd: {cmd}')
11
12              match (cmd):
13                  case 'next_batch':
14                      if (samples_counters[i] < samples_device):
15                          sendNextBatchOfDataToDevice(ser, i)
16                  case 'printing_model':
17                      print('Receiving model')
18                  case _:
19                      print('Unknown command')
20              s = s[cmdStart+cmdEnd:]
21              cmdStart = s.find('::')
22
23          print(f'Updated {ser.port} logs')
```

Funkcja *updateLogs* jest odpowiedzialna za proste dopisanie otrzymanych przez bufor danych do pliku z logami odpowiadającemu urządzeniu o danym porcie podczas danej sesji. Kod tej funkcji przedstawiono we fragmencie kodu 4.13. Wynikowa nazwa pliku *.log* zawiera dane pomagające jej kategoryzację oraz ułatwiające jej rozpoznanie. Na przykład dla mikrokontrolera dostępnego na porcie *COM7* logi testów na zestawie danych *housepricedata* zostały zapisane w pliku pod ścieżką:
logs/2022-09-03__test_housepricedata_17-39-54_COM7.log.

Kod 4.13. Funkcja updateLogs.

```
1 def updateLogs(s, port):
2     with open(f'logs/{session_start.strftime(f"%Y-%m-%d__      \
3             {session_name}_{data_filename}_%H-%M-%S_{port}")} \
4             .log', 'a') as f:
5         f.write(s)
```

Oprócz aktualizacji logów, serwer ma również za zadanie dostarczać dane treningowe do urządzeń. Odpowiedzialna za to jest funckja *sendNextBatchOfDataToDevice*, przedstawiona we fragmencie kodu 4.14. Bazując na globalnych licznikach z listy *samples_counters* wybiera kolejny zestaw danych do przekazania i następnie kolejno wpisuje w bufor bajty wartości atrybutów rekordów *Xtrain*, czeka na potwierdzenie ich odebrania od urządzenia w postaci pojedynczego znaku i wpisuje w bufor bajty wartości przynależności do kategorii tych rekordów *Ytrain*.

Kod 4.14. Funkcja sendNextBatchOfDataToDevice, przesyłająca kolejny pakiet danych do danego urządzenia.

```
def sendNextBatchOfDataToDevice(ser, idx):
    batch_data = Xtrain[idx][samples_counters[idx] : \
        samples_counters[idx]+samples_batch]
    batch_data_res = Ytrain[idx][samples_counters[idx] : \
        samples_counters[idx]+samples_batch]

    sent = ser.write(batch_data.tobytes())
    print(f'sent {sent} expected {batch_data.nbytes}')

    n = ser.read()

    sent = ser.write(batch_data_res.tobytes())
    print(f'sent {sent} expected {batch_data_res.nbytes}')

    samples_counters[idx] += samples_batch
    return
```

4.4. Program Arduino

Program wykorzystany na mikrokontrolerach napisano w języku C++. Jak każdy program Arduino składa się on z dwóch podstawowych funkcji przedstawionych we fragmencie kodu 4.15 - *setup* i *loop*. Pierwsza z tych funkcji ma za zadanie wykonać swój kod raz, na początku programu, w celu inicjalizacji odpowiednich ustawień. Druga funkcja, *loop*, to główna pętla programu, której kod wykonywany jest nieograniczoną ilość razy zaraz po wykonaniu *setup*.

Kod 4.15. Podstawowe funkcje programu Arduino.

```
void setup() {
  ...
}
void loop() {
  ...
}
```

Częścią przygotowań do wykonywania głównej pętli programu jest definicja stałych i zmiennych globalnych. Wśród nich najważniejszy jest obiekt klasy *NeuralNetwork* z wykorzystanej do uczenia maszynowego biblioteki *NeuralNetworks*. We fragmencie kodu 4.16 przedstawiono proces inicjalizacji tego obiektu dla przykładu sieci neuronowej z rysunku 5.1. W konstruktorze obiektu należało podać pożądany układ sieci, liczbę jej warstw i wybrane funkcje aktywacji.

Kod 4.16. Inicjalizacja obiektu NeuralNetwork sieci neuronowej.

```
const unsigned int layers[] = {10, 4, 4, 1};
```

```
2 byte activationFunctions[]  = {    1, 1, 0};
3 const unsigned int numLayers = 4;
4
5 NeuralNetwork mlNet = NeuralNetwork(layers, numLayers,
6                                      activationFunctions);
```

W celu wykorzystania różnych funkcji aktywacji w warstwach sieci, biblioteka wymaga zdefiniowania opcji *ACTIVATION__PER_LAYER* wraz z wybranymi funkcjami aktywacji przed dodaniem jej do programu. Definiowane kolejno funkcje we fragmencie kodu 4.17 tworzą listę, do której indeksów odwołano się w linii 41 fragmentu kodu 4.16.

Kod 4.17. Inicjalizacja biblioteki NeuralNetworks i jej opcji.

```
1 #define BINARY_CROSS_ENTROPY
2 #define ACTIVATION__PER_LAYER
3   #define Sigmoid // 0
4   #define ReLU    // 1
5
6 #include <NeuralNetwork.h>
```

Dodatkowo przed rozpoczęciem uczenia ważne jest również ustawienie szybkości uczenia modelu poprzez bezpośrednie nadpisanie wartości *mlNet.LearningRateOfWeights*. Następnie program urządzenia wpada w pętlę zaprezentowaną we fragmencie kodu 4.18. Pętla ta oczekuje na nadejście sygnału “start” z serwera.

Kod 4.18. Pętla oczekująca na sygnał “start” z serwera.

```
1 void setup() {
2   String read;
3   while (read != "start") {
4     read = Serial.readStringUntil('\n');
5   }
6   ...
```

Po przyjściu tego sygnału program wchodzi w główną pętlę, w której w pierwszej kolejności musi pobrać dane treningowe z serwera. Wykonuje to za pomocą funkcji *getNextBatchOfData*, której kod umieszczono we fragmencie 4.19. Funkcja ta odbiera bajty wysyłane w funkcji serwerowej 4.14. Najpierw zostaje wysłana komenda w postaci ciągu znaków “*::next_batch*” na bufor serial. Serwer interpretuje ją jako sygnał do rozpoczęcia przesyłania danych. Pętla *while* oczekuje na dostępność danych do odczytania w buforze. Po ich pojawieniu się, zostają one odczytane przez funkcję *Serial.readBytes* z przerwą na wpisanie znaku jako potwierdzenia otrzymania pierwszej części. Otrzymane dane przepisywane są z ciągu bajtów do zmiennych liczb zmiennoprzecinkowych w globalnych tablicach *batchData* oraz *batchDataResult*.

Kod 4.19. Funkcja pobierająca kolejną partię danych z serwera.

```
1 float batchData[BATCH_SAMPLE_COUNT]
```

```
2                 [BATCH_NUM_INPUT_VALUES];
3  float batchDataResult[BATCH_SAMPLE_COUNT]
4                       [BATCH_NUM_OUTPUT_VALUES];
5
6  void getNextBatchOfData() {
7    println("::next_batch");
8    const int batchDataSize = sizeof(batchData);
9    const int batchDataResSize =
10                sizeof(batchDataResult);
11   uint8_t batchDataBytes[batchDataSize];
12   uint8_t batchDataResBytes[batchDataResSize];
13
14   while (Serial.available() < 4) {}
15   Serial.readBytes(batchDataBytes, batchDataSize);
16   Serial.write("n");
17   memcpy(batchData, batchDataBytes, batchDataSize);
18
19   Serial.readBytes(batchDataResBytes, batchDataResSize);
20
21   int counter = 0;
22   for (int i=0; i < BATCH_SAMPLE_COUNT; ++i) {
23     for (int j=0; j < BATCH_NUM_OUTPUT_VALUES; ++j) {
24       batchDataResult[i][j] = batchDataResBytes[counter];
25       ++counter;
26     }
27   }
28   ...
29 }
```

Po wczytaniu danych, dana runda jest gotowa do przeprowadzenia treningu. Odpowiedzialna za to jest funkcja *train* przedstawiona we fragmencie kodu 4.20. Wykorzystuje ona funkcje klasy *NeuralNetwork*: *FeedForward* do przejścia wprzód i *BackProp* do wyliczenia i wdrożenia zmian uczących sieć. Pętla uczenia zostaje wykonana na całym dostępnym zestawie danych przez liczbę epok określoną przez *NUM_EPOCHS*.

Kod 4.20. Funkcja ucząca model na danych treningowych.

```
1  void train() {
2    for (int i=0; i < NUM_EPOCHS; ++i) {
3      for (int j=0; j < BATCH_SAMPLE_COUNT; ++j) {
4        mlNet.FeedForward(batchData[j]);
5        mlNet.BackProp(batchDataResult[j]);
```

Po ukończeniu treningu urządzenie podejmuje próbę odczytania danych z innych urządzeń w sieci mikrokontrolerów. Przy pomyślnej próbie dane zostają zapisane do tablicy globalnej *recValues*. Z racji zastosowania dwóch metod rozproszonego uczenia maszynowego - scalania wag i uśredniania gradientów - w następnym kroku pętli wykorzystana zostaje funkcja odpowiadająca wybranej w danym procesie uczenia metodzie.

Metodę scalania wag zaimplementowano w funkcji *mergeWithNetModel*, którą przedstawiono we fragmencie 4.21. Funkcja ta nadpisuje wagi modelu lokalnego średnią ważoną wagi dotychczasowej i wagi odczytanej iterując po warstwach i połączeniach pomiędzy neuronami. Wagi bias przechowywane są w obiekcie *NeuralNetwork* osobno od pozostałych wag, jako pojedyncze wartości dla każdej warstwy, które są parametrem funkcji aktywacji następnej warstwy. Ich uśrednianie odbywa się analogicznie do uśredniania wag, ale w zewnętrznej pętli funkcji.

Kod 4.21. Funkcja scalająca wagi odczytanego modelu z modelem lokalnym.

```
1  void mergeWithNetModel() {
2    int counter = 0;
3    for (int i=0; i < mlNet.numberOflayers; ++i) {
4      for (int j=0; j < layers[i+1]; ++j) {
5        for (int k=0; k < layers[i]; ++k) {
6          float weight = 0.618 * mlNet.layers[i].weights[j][k]
7                       + 0.382 * recValues[counter];
8          mlNet.layers[i].weights[j][k] = weight;
9          ++counter;
10         }
11     }
12     float bias;
13     memcpy(&bias, mlNet.layers[i].bias, sizeof(float));
14     bias = 0.618 * bias + 0.382 * recBiases[i];
15     memcpy(mlNet.layers[i].bias, &bias, sizeof(float));
16     mlNet.layers[i].bias = 0.618 * mlNet.layers[i].bias +
17                            0.382 * recBiases[i];
```

Alternatywą do powyższej metody jest uśrednianie gradientów, którego implementację zaprezentowano we fragmencie kodu 4.22. Gradienty lokalne zostają uśrednione średnią ważoną ze złotym podziałem z odczytanymi gradientami i wynik jest aplikowany do lokalnego modelu.

Kod 4.22. Funkcja aplikująca gradienty do lokalnego modelu.

```
1  void applyGrads() {
2    int counter = 0;
3    for (int i=0; i < mlNet.numberOflayers; ++i) {
4      for (int j=0; j < layers[i+1]; ++j) {
5        for (int k=0; k < layers[i]; ++k) {
6          float grad = mlNet.layers[i].weights[j][k] -
7                       prevWeights[counter];
8          float weight = prevWeights[counter] +
9                       (0.618 * grad + 0.382 * recValues[counter]);
10         mlNet.layers[i].weights[j][k] = weight;
11         ++counter;
12         ...
```

4.4.1. Komunikacja Bluetooth Low Energy

Komunikacja urządzeń przez Bluetooth Low Energy jest równie ważną częścią rozproszonego uczenia maszynowego jak samo uczenie. Zaimplementowano ją dzięki podstawowej bibliotece ArduinoBLE. Inicjalizację najważniejszych komponentów, czyli serwisu i charakterystyk z przekazywanymi danymi, przedstawiono we fragmencie kodu 4.23. Jako identyfikatory charakterystyk i serwisu wykorzystano wygenerowane losowo Universally Unique Identifier (UUID) w wersji 4. Rozmiary charakterystyk są modyfikowalne do 244 bajtów. Oznacza to, że w jednej charakterystyce mieści się 61 4-bajtowych liczb zmiennoprzecinkowych float lub 122 2-bajtowe liczby float o zmniejszonej precyzji.

Kod 4.23. Definicje zmiennych serwisu i charakterystyki BLE.

```
#include <ArduinoBLE.h>
...
const char* bleServiceUuid = "4e14b225-2e5b-48f5-945b-fa168de092bc";
const char* bleCharUuid = "2543aba0-68c2-4786-baba-d78a9bd9ceae";
const char* bleChar2Uuid = "7596fdd3-00f9-4e23-b0a0-1741905ddab1";

BLEService bleService(bleServiceUuid);
BLECharacteristic bleChar(bleCharUuid, BLERead | BLENotify,
                          sizeof(recValues), false);
BLECharacteristic biasesChar(bleChar2Uuid, BLERead | BLENotify,
                             sizeof(recBiases), false);
```

Podczas wykonywania funkcji *setup* wykonana zostaje również funkcja *init_ble*, która zajmuje się odpowiednim ustawieniem parametrów modułu BLE (fragment kodu 4.24). Funkcja ta czeka na włączenie modułu BLE, po czym ustawia nazwę urządzenia, dodaje charakterystykę *bleChar* do serwisu *bleService* i serwis do modułu BLE, po czym zaczyna go rozgłaszać.

Kod 4.24. Funkcja init_ble.

```
void init_ble() {
  if (!BLE.begin()) {
    println("BLE module failed to start");
    while(1);
  }
  BLE.setLocalName("Nano33 ML");

  bleService.addCharacteristic(bleChar);
  BLE.addService(bleService);
  BLE.setAdvertisedService(bleService);

  BLE.advertise();
}
```

W każdej rundzie po treningu aktualizowane są charakterystyki za pomocą funkcji *writeValue* biblioteki ArduinoBLE. We fragmencie kodu 4.25 przedstawiono wycinek funkcji

updateCharacteristics aktualizujący udostępniane dane. W załączniku 1. dołączono pełny kod tej funkcji dla metody scalania wag.

Kod 4.25. Fragment funkcji updateCharacteristics.

```
1 void updateCharacteristics() {
2   ...
3   bleChar.writeValue(charBytes, weightsSize);
4   biasesChar.writeValue(biasesBytes, biasesSize);
5 }
```

Po drugiej stronie połączenia z urządzeniem wystawiającym dane jest urządzenie je odczytujące. Dany mikrokontroler łączy się z innym przez BLE podczas wykonania funkcji *connectToPeripheral*, przedstawionej we fragmencie kodu 4.26. W ramach uproszczenia protokołu komunikacyjnego, urządzenia łączą się ze sobą za pomocą znanych adresów MAC. W przypadku gdy z jakiegoś powodu drugie urządzenie jest niewykrywalne, skan kończy się z niepowodzeniem po upłynięciu czasu określonego przez stałą *SCAN_TIMEOUT*, która domyślnie wynosi 10000 ms, czyli 10 sekund. Przy pomyślnym odnalezieniu urządzenia skan zostaje zakończony i wywołana zostaje funkcja *controlPeripheral*.

Kod 4.26. Funkcja connectToPeripheral.

```
1  bool connectToPeripheral(String devAddress) {
2    BLEDevice peripheral;
3    unsigned long scanStart = millis();
4    do {
5      BLE.scanForAddress(devAddress);
6      peripheral = BLE.available();
7    } while (!peripheral && (millis() - scanStart < SCAN_TIMEOUT));
8    if (peripheral) {
9      BLE.stopScan();
10     return controlPeripheral(peripheral);
11   }
12   BLE.stopScan();
13   return false;
14 }
```

Funkcja *controlPeripheral* odpowiedzialna jest za odkrycie atrybutów udostępnianych przez drugie urządzenie i następnie odczytanie ciągów bajtów z odpowiednich charakterystyk, co przedstawia fragment kodu 4.27. Pełny kod tej funkcji umieszczono w załączniku 2.

Kod 4.27. Fragment funkcji controlPeripheral.

```
1 bool controlPeripheral(BLEDevice peripheral) {
2   ...
3     const int weightsSize = sizeof(weightValues);
4     uint8_t weightsBytes[weightsSize];
5     recWeightsChar.readValue(weightsBytes, weightsSize);
```

```
6
7       const int biasesSize = sizeof(recBiases);
8       uint8_t biasesBytes[biasesSize];
9       recBiasesChar.readValue(biasesBytes, biasesSize);
10
11      memcpy(recValues, weightsBytes, sizeof(recValues));
12      memcpy(recBiases, biasesBytes, biasesSize);
13    ...
```

4.4.2. Testowanie modeli

Modele trenowane na urządzeniach Arduino testowano w programie Python z użyciem biblioteki Keras, aby zapewnić im takie samo środowisko testowe jakie miały modele z symulacji. W celu przeprowadzenia tych testów, program najpierw wczytuje dane testowe analogicznie do fragmentu kodu 4.2.

Następnie wczytuje model lub modele z plików *.log* w postaci zwykłych ciągów znaków String. Modele przesyłane z urządzenia Arduino do serwera przez bufor serial otrzymywane są za pomocą wywołania funkcji *NeuralNetwork::print()* i mają postać przedstawioną we fragmencie kodu 4.28, w którym widnieje przykład modelu zbudowanego na podstawie sieci z rysunku 5.1 o układzie {10, 4, 4, 1}. Sieć opisano w kolejnym rozdziale pracy.

Kod 4.28. Model przesłany z mikrokontrolera.

```
1  ----------------------
2  10 4| bias:0.97
3  1   W:-0.7543511   W: 0.8459654   W:-0.1379976   W: 0.7710103 ...
4  2   W:-0.0614348   W:-0.8436971   W: 0.8974885   W:-0.2987273 ...
5  3   W: 0.4877181   W: 0.3664785   W: 0.6124614   W: 0.0345318 ...
6  4   W:-0.3794431   W:-0.7193038   W:-0.3164640   W:-0.4050426 ...
7  ----------------------
8  4 4| bias:1.00
9  1   W:-0.8495268   W: 1.4702393   W: 0.6771587   W:-0.6375570
10 2   W: 0.6880698   W:-1.4531732   W:-0.3050147   W: 0.1725868
11 3   W: 0.7695336   W:-0.7812655   W:-0.3737841   W:-0.2785285
12 4   W: 0.1151042   W: 0.8331513   W:-0.2586055   W: 0.4443044
13 ----------------------
14 4 1| bias:0.35
15 1   W:-1.7498884   W: 0.8186014   W: 0.4874596   W:-0.9154620
16 ----------------------
```

Pierwsza linia separatorów określa początek wypisania modelu. Następnie widoczne są trzy bloki oddzielone kolejnymi liniami separatorów (linie 7 i 13). Każdy z bloków opisuje stan danej warstwy. Blok rozpoczyna liczba neuronów w danej warstwie, zaczynając od warstwy wejściowej w linii 2. Druga liczba oznacza ilość neuronów w kolejnej warstwie. Liczba *bias* to wartość biasu danej warstwy. W kolejnych liniach wypisane zostały wszystkie wagi pomiędzy daną warstwą i warstwą kolejną. Pozostałe bloki analogicznie opisują stan kolejnych warstw.

Każdy z testowanych modeli wymaga inicjalizacji i kompilacji modelu Keras o odpowiedniej strukturze. Następnie wagi tego modelu zostają nadpisane wagami modelu uzyskanego na urządzeniu Arduino. Do parsowania i wczytania go do modelu Keras służy zaimplementowana funkcja *readModel*, przedstawiona w załączniku 3. W przypadku wystąpienia błędu w postaci wartości *ovf* lub *nan* funkcja ta zeruje daną wagę. Proces wczytania modelu do programu oraz testowania na zestawie testowym przedstawiono we fragmencie kodu 4.29.

Kod 4.29. Operacje wczytujące i testujące model uzyskany na urządzeniu Arduino do modelu Keras.

```
1 model1 = init_model()
2 model1.compile(loss='binary_crossentropy', \
3                optimizer=optim, metrics=['accuracy'])
4 readModel(model1, s1)
5 _, accuracy1 = model1.evaluate(Xtest, Ytest)
```

5. Badania

W celu zbadania efektywności rozproszonego uczenia maszynowego w sieci mikrokontrolerów, przeprowadzono symulacje z wykorzystaniem metod scalania wag oraz scalania gradientów. Do badań wykorzystano dwa zbiory danych - pierwszy klasyfikuje przynależność rekordów do dwóch kategorii, a drugi do dziesięciu. Dzięki różnicom pomiędzy tymi zbiorami uzyskano różnorodne wyniki. Następnie, po przeprowadzeniu symulacji, przetestowano rzeczywistą sieć mikrokontrolerów, złożoną z trzech urządzeń Arduino Nano 33 BLE Sense.

5.1. Rozproszone uczenie maszynowe

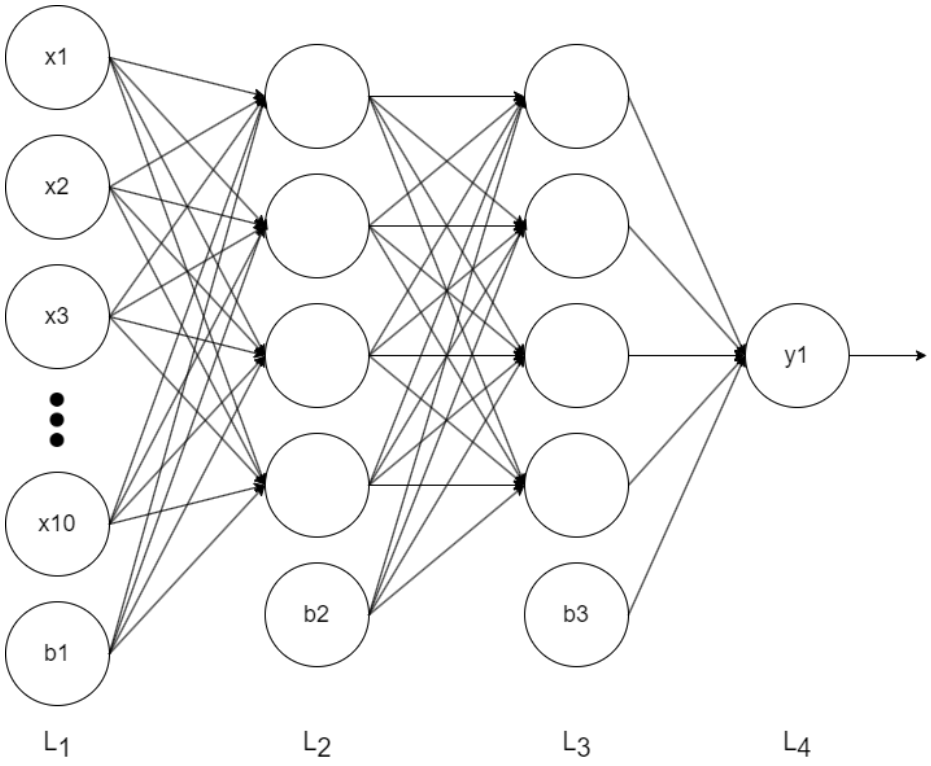
W celu zbadania czy rozproszone uczenie maszynowe ze scalaniem wag modeli oraz z uśrednianiem gradientów dają pożądane efekty, najpierw przeprowadzono symulacje w środowisku z bibliotekami TensorFlow i Keras w języku Python. Do uczenia maszynowego wykorzystano optymalizatory Stochastic Gradient Descent (SGD) i Adam, aby porównać w jakim stopniu zastosowane metody są z nimi współpracują i czy dany optymalizator jest lepszy dla danej metody. W przeprowadzonych eksperymentach wykorzystano popularny zbiór danych do rozpoznawania cyfr zapisanych ręcznie MNIST oraz mniej obszerny zbiór danych firmy Zillow, klasyfikujący ceny nieruchomości na podstawie różnych atrybutów.

5.1.1. Klasyfikacja cen domów - scalanie wag

Do pierwszego badania wykorzystano zbiór danych klasyfikujący przynależność nieruchomości do grupy poniżej lub powyżej mediany ceny rynkowej na podstawie określonych atrybutów domu. Cały zestaw (1461 rekordów) podzielono na zestaw treningowy o wielkości 900 rekordów i zestaw testowy o wielkości 400 rekordów. W dalszej części badań w celu uczenia 3 modeli zestaw treningowy podzielono na 3 mniejsze zestawy po 300 rekordów każdy.

W badaniu zastosowano w pełni połączoną sieć neuronową o układzie {10, 4, 4, 1} zaprezentowanym na rysunku 5.1. Jako funkcje aktywacji w sieci wykorzystano funkcję ReLU w warstwach L_2 i L_3 oraz funkcję Sigmoid w warstwie L_4. Dodatkowo warstwy L_1, L_2 i L_3 poszerzono o neurony bias, tj. dostarczające składowej stałej, b_1, b_2 i b_3 o wartości 1, odpowiednio po 1 na warstwę. W sumie sieć składa się z 60 wag normalnych oraz 9 wag dla składowej stałej. Jako funkcję straty zastosowano binary crossentropy.

Na początek, w ramach testu kontrolnego, zbadano precyzję sieci nauczonej za pomocą biblioteki Keras w sposób tradycyjny, czyli na pełnym zbiorze treningowym (900 rekordów). Do treningu wykorzystano ustawienia 10 epok, o rozmiarze batch_size równym 20. Dla porównania użyto dwóch optymalizatorów z biblioteki Keras: SGD o wartości parametru określającego tempo uczenia learningRate równej 0.1 i Adam o takiej samej wartości parametru learningRate. Parametr ten w obu przypadkach ma dość wysoką wartość w porównaniu do powszechnych zastosowań, ale biorąc pod uwagę relatywnie



Rysunek 5.1. Schemat modelu sieci neuronowej zastosowanego w badaniach z danymi o cenach domów

małą objętość danych treningowych, przy takiej wartości zapewniał najlepszą precyzję. Trafność predykcji sieci względem zbioru testowego wyniosła 86.5% z wykorzystaniem optymalizatora SGD oraz 86.75% z optymalizatorem Adam. Wyniki te są stosunkowo dobre biorąc pod uwagę znaczącą redukcję objętości oryginalnego zestawu danych, co wpłynęło na dokładność wykorzystanych atrybutów.

Następnie przeprowadzono symulacje rozproszenia uczenia maszynowego poprzez podzielenie treningu na trzy modele, odpowiadające trzem urządzeniom. Do procesu treningu wykorzystano wcześniej wspomniane 3 mniejsze zestawy danych po 300 rekordów każdy, odpowiednio po 1 zestawie na model, aby zachować rozłączność zbiorów. Wyniki nauczenia tych trzech modeli osobno z wykorzystaniem optymalizatora SGD przedstawiono w tabeli 5.1. W badaniu zastosowano parametr learningRate zwiększony do 0.3 i batch_size zmniejszony do 10. Dzięki dostosowaniu tych parametrów do zmniejszonych zbiorów treningowych, celność modeli wypadła niewiele gorzej od testu kontrolnego na spójnym zbiorze.

Tabela 5.1. Wyniki uczenia w rundach na zestawie cen domów, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Model 1	Model 2	Model 3
85.50	84.75	85.75

Analogicznie wykonano podobny test dla optymalizatora Adam. Tym razem jednak wyniki podobne do testu kontrolnego, przedstawione w tabeli 5.2, uzyskano poprzez zmniejszenie wartości parametru learningRate do 0.05. W przypadku modelu 2 uzyskano wynik lepszy o 2% od optymalizatora SGD.

Tabela 5.2. Wyniki uczenia w rundach na zestawie cen domów, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Model 1	Model 2	Model 3
85.50	86.25	86.00

Mając dane z testów kontrolnych, przeprowadzono eksperymenty mające na celu połączenie zdolności oddzielnie nauczonych modeli, wykorzystując metody rozproszonego uczenia maszynowego.

W pierwszym eksperymencie założono "doskonałe" warunki symulacji, czyli z synchronicznym procesem scalania wag. Całość programu podzielono na rundy, podczas których najpierw każdy z modeli przechodzi trening na kolejnej partii danych ze swojego zestawu, który podzielono na równe części dla każdej rundy, przez określoną liczbę epok. Następnie wagi wszystkich modeli są uśredniane ze swoimi odpowiednikami w innych modelach i uzyskane wartości są przypisywane jednakowo do każdego modelu. Po określonej liczbie rund uczenie kończy się i wynikowe modele są jednakowe. Wzór na otrzymaną w danej rundzie wagę określa równanie:

$$w_{sc} = \frac{w_1 + w_2 + w_3}{3}$$

W tabeli 5.3 przedstawiono wyniki symulacji przeprowadzonej dla 4 rund scalania wag, gdzie w każdej rundzie każdy z trzech modeli poddano treningowi na 25% (75 rekordach) przypisanego do niego zestawu danych treningowych przez 10 epok. W każdej rundzie obliczono trafność predykcji modeli na zestawie testowym po treningu na kolejnych partiach danych, a następnie scalono wagi i znowu przetestowano trafność predykcji. W danym eksperymencie wykorzystano optymalizator SGD biblioteki Keras z parametrem learningRate (szybkość uczenia) zwiększonym do wartości 0.3 oraz parametrem batch_size zmniejszonym do 10. Zastosowane parametry zapewniły lepsze wyniki od poprzednich wartości parametrów w tym scenariuszu, ze względu na podzielenie danych treningowych na 3 modele i 4 rundy.

Jak widać w tabeli 5.3 w pierwszej rundzie trafność predykcji nie jest zbyt dobra dla modeli 2 i 3, a model 1 osiąga zaledwie 51.25% co oznacza, że nie nauczył się dobrze. Z kolei model ze scalonymi wagami osiągnął trafność predykcji na poziomie zaledwie 49.25%. Oznacza to, że niektóre lub wszystkie z trzech modeli miały rozbieżne wagi i wynikowy model praktycznie zgadywał predykcje. Sytuacja znacznie poprawiła się od drugiej rundy, gdzie wejściowy model to model scalony z pierwszej rundy, dzięki czemu proces uczenia ma spójny punkt startowy. Precyzja predykcji spada jedynie dla modelu

Tabela 5.3. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	51.25	63.50	67.00	49.25
2	86.75	82.75	85.00	86.25
3	85.00	83.50	84.50	86.50
4	84.00	84.00	71.50	85.50

3 w ostatniej rundzie, ale wynikowy model ze scalonymi wagami osiąga nawet większą precyzję niż jego składowe modele.

Z wyjątkiem pierwszej rundy, gdzie trzy uczone modele miały różne, losowe wagi wejściowe, uśrednianie wag pomogło osiągnąć precyzję predykcji lepszą od najgorszego modelu w danej rundzie, a w przypadku rund 3 i 4 lepszą od wszystkich trzech modeli.

W tabeli 5.4 przedstawiono wyniki analogicznego eksperymentu. Tym razem wykorzystano optymalizator Adam biblioteki Keras z parametrem learningRate o wartości równej 0.05. W porównaniu do optymalizatora SGD, Adam osiąga minimalnie lepsze i bardziej spójne wyniki w poszczególnych rundach.

Tabela 5.4. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	82.75	83.25	82.75	49.25
2	80.25	82.75	81.25	85.75
3	81.50	85.00	85.50	86.00
4	85.75	83.75	84.25	86.25

Zbadano również wpływ zmiany ilości rund na wyniki. W tabelach 5.5 oraz 5.6 przedstawiono podzielenie procesu uczenia na 2 rundy zamiast 4, odpowiednio z zastosowaniem optymalizatorów SGD i Adam. W tym przypadku, ze względu na podział na 2 rundy, każdy z modeli w każdej rundzie wykonał trening na zbiorze 150 rekordów. Tak jak w poprzednim teście, pierwsza runda posłużyła jako naprowadzenie modeli do dążenia do tego samego minimum funkcji straty, dzięki czemu w drugiej rundzie uzyskano dobry model wynikowy, kończąc trening. Z kolei optymalizator Adam zarówno w pierwszej jak i drugiej rundzie wytrenował 3 rozbieżne modele, przez co w obu z nich uzyskał model zgadujący, czyli z celnością około 50%. Stało się tak prawdopodobnie ze względu na większą ilość danych w rundach, co umożliwiło poszczególnym modelom na "ucieczkę" od wspólnego modelu otrzymanego po pierwszej rundzie.

O ile zmniejszenie liczby rund w przypadku optymalizatora Adam dało niepożądane wyniki, tak w przypadku SGD wypadło nawet niewiele lepiej. W dodatku w zastosowaniu na rzeczywistych mikrokontrolerach zmniejszyłoby nakład na komunikację pomiędzy urządzeniami. Niestety przedstawiony scenariusz zakłada idealne warunki, czyli iden-

Tabela 5.5. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, 2 rundy, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	84.00	83.00	79.00	49.25
2	85.00	81.75	87.00	86.75

Tabela 5.6. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, 2 rundy, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	85.00	84.75	83.75	49.25
2	86.00	86.25	67.75	50.75

tyczny czas rozpoczęcia uczenia i synchroniczną wymianę wag, co w realnych warunkach mogłoby być trudne do uzyskania. Dodatkowo zakłada doskonałą wymianę wag pomiędzy urządzeniami, co również nie zawsze jest osiągalne z prawdziwymi urządzeniami - jak opisano w dalszym podrozdziale - dlatego scenariusza 2 rund uczenia nie wykorzystano do dalszych badań.

Aby zbadać odwrotną sytuację, zwiększono liczbę rund do 10. Wyniki przedstawiono w tabeli 5.7 dla optymalizatora SGD oraz w tabeli 5.8 dla optymalizatora Adam. Przez podzielenie uczenia na 10 rund, liczba rekordów danych treningowych w każdej rundzie zmniejszyła się do 30 na model. Jest to bardzo mała ilość danych, ale dzięki temu modele uczone za pomocą optymalizatora Adam nie miały jak oddalić się od wspólnego minimum tak jak w poprzednim eksperymencie z wykorzystaniem jedynie 2 rund. Z wyników trafności predykcji modeli można wywnioskować, że modele SGD szybko znalazły wspólne minimum i oscylowały wokół niego, podczas gdy modele Adam zaczęły się zbiegać dopiero w czwartej rundzie po czym model scalony stabilnie dążył do minimum funkcji straty.

Tabela 5.7. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, 10 rund, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	52.50	60.00	71.75	49.25
2	77.75	81.00	60.00	86.50
3	86.25	85.50	80.50	85.25
4	77.75	82.25	83.00	84.50
5	83.25	65.25	85.50	85.50
6	82.75	80.50	82.75	85.75
7	82.75	85.25	71.50	87.00
8	59.25	86.00	85.50	83.25
9	84.25	83.00	84.75	85.25
10	79.00	87.00	85.75	85.50

Tabela 5.8. Wyniki symulacji komunikacji synchronicznej na zestawie cen domów, 10 rund, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	67.25	49.25	83.75	49.25
2	49.25	50.75	50.75	50.75
3	50.75	50.75	82.25	59.75
4	84.25	82.00	79.00	84.25
5	87.00	84.25	85.25	85.25
6	82.25	76.50	86.25	85.75
7	83.50	84.50	83.75	85.75
8	85.25	86.00	85.50	86.50
9	86.50	85.75	85.75	86.75
10	85.75	83.50	86.25	86.75

Zwiększenie liczby rund udowodniło, że zmniejszenie danych treningowych w danej rundzie przy zwiększeniu częstotliwości wymiany wag pomiędzy modelami daje wyniki podobne do testów dla 4 rund, dlatego o ile może mieć to pozytywny efekt pod względem stabilności w środowisku komunikacji mikrokontrolerów, gdzie wymiana danych nie zawsze się powodzi, o tyle w środowisku symulacyjnym nie ma takich obaw, dlatego kolejne symulacje przeprowadzono dla 4 rund uczenia z uwagi na lepszą czytelność tabeli.

Poprzednie symulacje zakładają warunki, w których scalanie wag na koniec każdej rundy odbywa się synchronicznie, co może nie być łatwe lub nawet praktyczne w środowisku sieci mikrokontrolerów, gdzie zamysłem jest samodzielność każdego z urządzeń. Z tego powodu przeprowadzono kolejne symulacje, tym razem z inną metodą scalania wag. Wagi danego modelu w danej rundzie są nadpisywane średnią ważoną jego dotychczasowych wag oraz wag jednego innego modelu według poniższego równania, które przedstawia złoty podział. Metoda ta została zastosowana w celu trochę wierniejszego odwzorowania komunikacji asynchronicznej pomiędzy urządzeniami. Proces scalania wag w rundzie, tak jak w poprzednim eksperymencie, wciąż jest synchroniczny i zakłada dostępność pozostałych modeli oraz możliwość odczytania ich wag, co wciąż nie jest całkowicie wiernym odwzorowaniem realnej sieci urządzeń, w której dane urządzenie może nie mieć dostępu do wszystkich innych urządzeń w momencie chęci scalenia wag, ale uwzględnia dostępność tylko jednego innego urządzenia naraz.

$$w_1 = 0.618 * w_1 + 0.382 * w_2$$

Proces scalania wag odbywa się zgodnie z algorytmem przedstawionym we fragmencie kodu 5.1 dla przykładu modelu 1, którego wagi zostają nadpisane średnią ważoną z wagami modelu 2 zgodnie z powyższym równaniem i następnie analogicznie wynikowe wagi zostają nadpisane kolejną średnią ważoną, tym razem z wagami modelu 3. Taki sam proces przechodzą pozostałe modele.

Kod 5.1. Przykład obliczenia nowych wag dla modelu 1.

```
1  weights1[i] = 0.618*weights1[i] + 0.382*weights2[i]
2  weights1[i] = 0.618*weights1[i] + 0.382*weights3[i]
```

Wyniki przeprowadzonego eksperymentu dla optymalizatora SGD z biblioteki Keras o learningRate = 0.3 i batch_size = 10 przedstawiono w tabeli 5.9. Faza treningu w pierwszej rundzie odbyła się bez zmian w odniesieniu do pierwszego testu komunikacji synchronicznej (tabela 5.3), więc poszczególne wyniki były w niej takie same. Trafność scalonych modeli również nie odchyliła się od ich odpowiednika z analogicznej symulacji, ze względu na różnice pomiędzy ich wagami. Kolejne rundy osiągnęły podobne poziomy trafności dla wszystkich scalonych modeli, pomimo innej metody scalania wag.

Tabela 5.9. Wyniki symulacji komunikacji asynchronicznej na zestawie cen domów, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	51.25	63.50	67.00	49.25	49.25	49.25
2	85.75	83.00	84.50	80.75	82.00	83.50
3	84.75	81.25	84.75	85.50	84.75	85.00
4	83.75	86.50	71.50	86.00	86.00	85.75

Na rysunku 5.2 przedstawiono wzrost przewagi precyzji modeli po ich scaleniu nad precyzją modeli bezpośrednio po treningu w kolejnych rundach na podstawie wyników z tabeli 5.9.

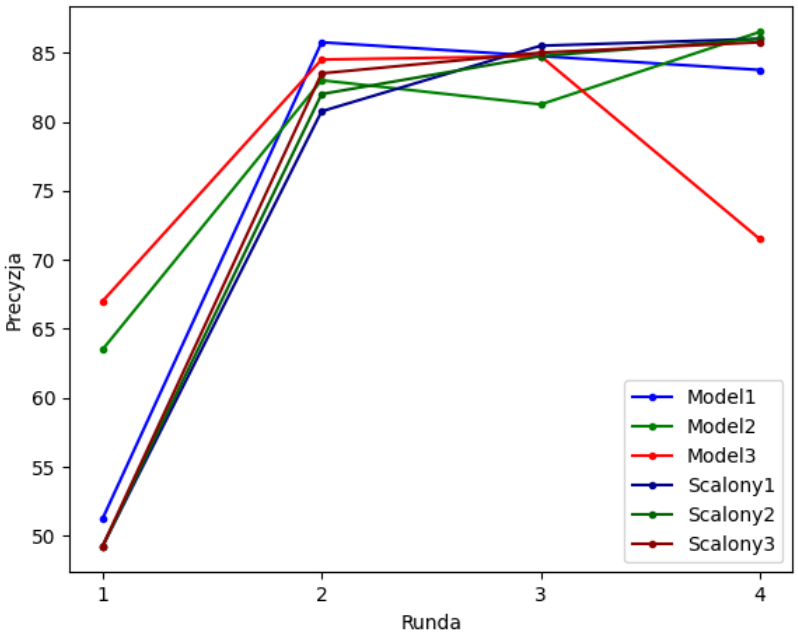
W tabeli 5.10 zaprezentowano wyniki symulacji z taką samą metodą scalania wag dla optymalizatora Adam z biblioteki Keras o learningRate = 0.05 i batch_size = 20, analogicznie do poprzedniego eksperymentu. Ogólna trafność modeli scalonych jest bardzo podobna do poprzedniej symulacji, z wykorzystaniem optymalizatora SGD.

Tabela 5.10. Wyniki symulacji komunikacji asynchronicznej na zestawie cen domów, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	82.75	83.25	82.75	49.25	49.25	49.25
2	85.75	82.50	81.00	85.00	84.50	74.50
3	83.50	85.00	83.75	85.50	86.00	86.00
4	85.50	87.00	82.25	86.25	87.00	86.50

5.1.2. Klasyfikacja cen domów - uśrednianie gradientów

Innym sposobem na zastosowanie rozproszonego uczenia maszynowego jest uśrednianie gradientów. Podobnie jak w metodzie scalania wag, cały proces uczenia podzielono na rundy, ale w tym przypadku zamiast bezpośredniego przekazywania wag, metoda ta



Rysunek 5.2. Wykres precyzji w rundach przedstawionych w tabeli 5.9

polega na obliczeniu przyrostu każdej z wag danego modelu w danej rundzie i dodaniu go do odpowiadającej jemu wagi w innym modelu.

Podobnie do poprzedniej metody - wyniki, przedstawione w tabeli 5.11, pokazały że przy pojedynczej rundzie treningu w przypadku optymalizatora SGD w przypadkach modeli 1 i 2 zastosowanie metody pogorszyło wyniki, a w przypadku modelu 3 zepsuło nauczony model. Z kolei wyniki dla optymalizatora Adam, z tabeli 5.12 pokazały, że po wdrożeniu uśrednionych gradientów wszystkie modele zostały rozstrojone.

Tabela 5.11. Wyniki symulacji wymiany gradientów na zestawie cen domów, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Model			Model Scalony		
1	2	3	1	2	3
85.50	84.75	85.75	81.00	83.25	50.75

Tabela 5.12. Wyniki symulacji wymiany gradientów na zestawie cen domów, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Model			Model Scalony		
1	2	3	1	2	3
85.50	86.25	86.00	49.25	49.25	49.25

Następnie zwiększono liczbę rund wymiany gradientów. Dzięki temu uzyskano o wiele stabilniejsze wyniki. Do badania o 6 rundach treningu wykorzystano optymalizatory

z niezmiennymi parametrami learningRate oraz batch_size. Wyniki przedstawiono w tabelach 5.13 oraz 5.14. Analogicznie do metody scalania wag, pierwsza runda uczenia została wykorzystana głównie w celu skalibrowania mniej lub bardziej rozbieżnych modeli o losowych wagach startowych, przy czym mimo tego okazjonalnie dany model kończący pierwszą rundę uzyskał zaskakującą precyzję - na przykład model 3 w przypadku SGD. Modele utrzymały wysoką precyzję od rundy 2 dla SGD i od rundy 3 dla Adam, przy czym dla obu optymalizatorów uzyskano zbieżne modele końcowe o precyzji podobnej do i nawet większej od badania kontrolnego (SGD - 86.5%, Adam - 86.75%).

Tabela 5.13. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator SGD (learningRate = 0.3, batch_size = 10) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	51.50	79.00	78.50	71.75	63.25	82.25
2	86.00	83.00	84.25	81.75	82.75	84.25
3	86.75	82.50	84.75	86.50	82.25	85.50
4	81.00	79.25	76.25	81.75	80.75	80.75
5	85.00	86.00	86.25	86.50	85.50	86.75
6	86.25	87.25	88.25	85.75	86.00	88.75

Tabela 5.14. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator Adam (learningRate = 0.05, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	53.00	71.50	83.00	63.25	49.25	59.50
2	82.25	84.25	84.75	67.25	85.50	80.75
3	85.50	83.25	86.00	86.25	81.50	82.75
4	83.75	86.25	86.50	82.00	86.25	82.75
5	85.75	86.00	87.50	86.75	85.00	87.25
6	86.75	84.75	85.00	85.75	85.50	88.00

Biorąc pod uwagę ograniczenia Bluetooth Low Energy związane z jego przepustowością danych, zbadano również efekty zmniejszenia precyzji przekazywanych danych z 4-bajtowych float32 do 2-bajtowych float16. We wszystkich przeprowadzonych takim sposobem badaniach na zestawie cen domów żadna z celności predykcji modeli w poszczególnych rundach nie uległa zmianie. Oznacza to, że możliwe jest zmniejszenie obciążenia komunikacji o połowę potencjalnie bez utraty na dokładności predykcji modeli.

Oprócz zmniejszenia precyzji przekazywanych gradientów, zbadano także zastosowanie parametru odcięcia gradientu gradThreshold, czyli wartości, poniżej której gradienty zerowano. Przetworzone w taki sposób gradienty symulują komunikowanie jedynie istotnych zmian w wagach modelu. W taki sposób uzyskano jeszcze mniejsze obciążenie symulowanego kanału komunikacji.

Dla optymalizatora SGD nawet dla niewielkiej wartości gradThreshold równej 0.01 w przypadku modelu 3 ilość odciętych gradientów wyniosła 22.22%. W sumie podczas całego procesu uczenia wyzerowanych zostało 12.16% wszystkich gradientów. Mimo to precyzja predykcji utrzymała się na podobnych poziomach lub nawet wzrosła. Model 3 osiągnął 88.25%, czyli 0.25% więcej niż w teście bez zastosowania parametru odcięcia gradientów. Wyniki symulacji zaprezentowano w tabeli 5.15.

Tabela 5.15. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator SGD (learningRate = 0.3, batch_size = 10, gradThreshold = 0.01) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	51.50	79.00	78.50	71.50	63.25	82.25
2	84.25	83.00	84.25	80.75	82.75	84.25
3	86.75	82.50	84.75	87.00	82.25	85.50
4	81.00	79.25	76.25	81.75	80.75	80.75
5	85.00	86.00	86.25	86.50	85.75	86.75
6	86.50	86.75	88.25	85.50	86.50	88.25

Z kolei w tabeli 5.16 przedstawiono wyniki analogicznego badania z wykorzystaniem optymalizatora Adam. Trafność predykcji w tym eksperymencie również niewiele różni się od poprzedniej symulacji, ale tym razem odcięto znacznie więcej - aż 29.47% - gradientów o wartościach poniżej gradThreshold.

Tabela 5.16. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator Adam (learningRate = 0.05, batch_size = 20, gradThreshold = 0.01) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	53.00	71.50	83.00	63.25	49.25	59.50
2	82.50	84.25	84.50	67.25	85.50	80.75
3	85.25	83.25	86.00	86.25	81.25	82.75
4	83.75	86.25	86.50	81.50	86.25	82.75
5	85.75	86.25	87.00	86.75	85.25	87.25
6	86.00	84.75	88.00	87.00	85.25	88.25

Następnie zwiększono wartość parametru gradThreshold do 0.1, czyli dziesięciokrotnie. Dzięki temu jedynie na prawdę znaczące gradienty zostały przekazane pomiędzy modelami. W tabeli 5.17 przedstawiono wyniki tego badania. Średnia precyzja wynikowych modeli spadła o 0.6%, przy czym odcięto aż 55.56% gradientów. Strata trafności predykcji była stosunkowo niewielka w porównaniu z ilością odciętych gradientów.

W tabeli 5.18 przedstawiono wyniki analogicznego badania dla optymalizatora Adam. Tym razem średnia wynikowych modeli wzrosła o średnio 0.6% przy odcięciu 51.85% gradientów poniżej wartości gradThreshold. Oznacza to, że oba optymalizatory utrzymują

Tabela 5.17. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator SGD (learningRate = 0.3, batch_size = 10, gradThreshold = 0.1) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	51.50	79.00	78.50	69.50	66.75	82.00
2	85.50	82.25	85.00	85.25	82.75	86.00
3	86.50	82.50	84.75	86.50	82.50	84.75
4	82.25	78.75	81.50	84.00	80.75	82.50
5	84.50	85.75	86.00	85.50	84.25	85.50
6	86.00	87.00	88.75	84.75	85.75	88.00

podobne precyzje wynikowych modeli nawet przy odrzuceniu ponad połowy gradientów, ponieważ komunikowano jedynie znaczące zmiany w modelach.

Tabela 5.18. Wyniki symulacji wymiany gradientów na zestawie cen domów, 6 rund, optymalizator Adam (learningRate = 0.05, batch_size = 20, gradThreshold = 0.1) (%)

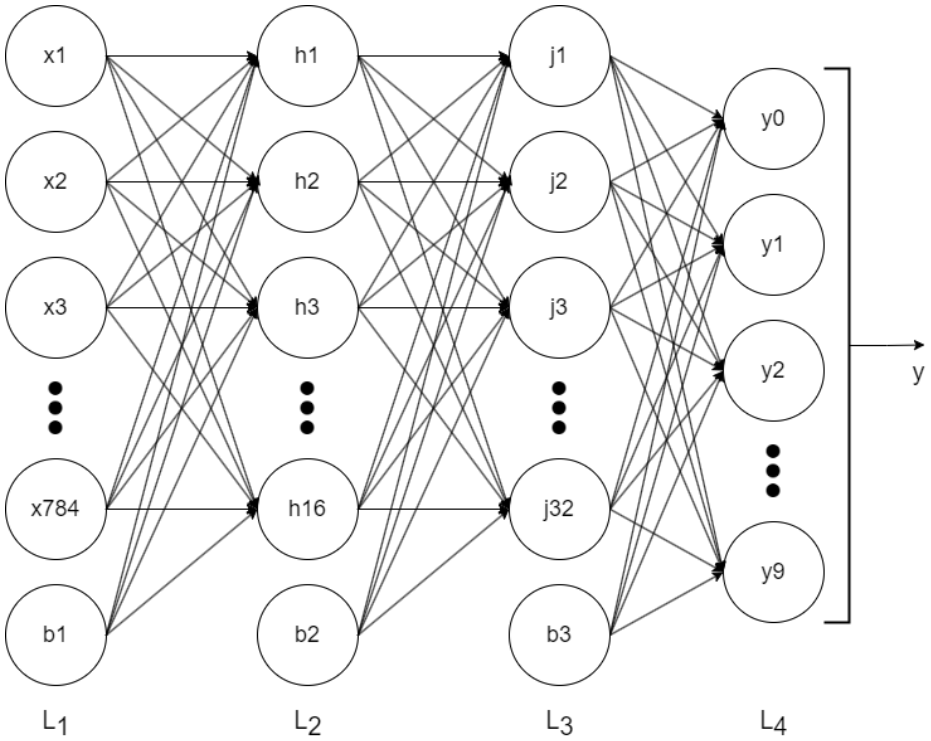
Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	53.00	71.50	83.00	60.75	49.25	60.00
2	83.25	84.50	83.75	65.50	85.25	83.00
3	85.25	83.00	85.50	86.25	81.75	81.50
4	82.00	85.25	82.25	80.75	86.25	75.75
5	86.00	86.00	89.25	86.75	83.50	87.25
6	86.25	86.25	86.75	87.25	88.00	87.00

Dzięki metodzie obcięcia mniej znaczących gradientów w połączeniu ze zmniejszeniem precyzji samych komunikowanych ich wartości z 4-bajtowego float32 na 2-bajtowy float16 udowodniono, że istnieje możliwość znacznej redukcji obciążenia kanału komunikacyjnego pomiędzy modelami (efektywnie urządzeniami) przy utrzymaniu podobnych precyzji modeli. Początkowo z danego modelu przekazywano wszystkie 60 gradientów, odpowiadających 60 wagom, w postaci 4-bajtowych liczb float32. Rozmiar takiej wiadomości wynosił 60*4=240 bajtów. Po zmianie typu przekazywanych wartości na 2-bajtowe liczby float16 objętość takiej wiadomości zmniejszono do 60*2=120 bajtów. Następnie dzięki zastosowaniu metody obcięcia gradientów istnieje możliwość jeszcze większego zmniejszenia rozmiaru tej wiadomości, ale niekoniecznie jest to zawsze opłacalne, ponieważ korzystając z Bluetooth Low Energy należy zakodować wiadomość, aby ją przesłać i następnie odkodować, aby ją odczytać. Oznacza to, że chcąc skorzystać z metody obcięcia gradientów, należy każdą z wartości float16 sparować z indeksem, aby druga strona była w stanie poprawnie ją przydzielić. Wykorzystany model ma mniej niż 128 wag, więc możliwym indeksem jest 1-bajtowa wartość Integer bez znaku. Uzyskany w taki sposób ciąg bajtów składa się z 3-bajtowych sekcji przeznaczonych na kolejne gradienty.

5.1.3. Klasyfikacja cyfr MNIST - scalanie wag

W ramach zweryfikowania wykonanych badań, analogiczne eksperymenty wykonano na zbiorze danych do klasyfikacji cyfr MNIST.

Do uczenia maszynowego wykorzystano znacznie większą, w porównaniu do badań na poprzednim zbiorze danych, sieć neuronową przedstawioną na rysunku 5.3. Sieć ta składa się z czterech, w pełni połączonych, warstw o układzie {784, 16, 32, 10}. W warstwach L_2 i L_3 wykorzystano funkcję aktywacji ReLU, a w warstwie wyjściowej L_4 funkcję aktywacji Softmax. Dodatkowo, jak w poprzednim eksperymencie, warstwy wejściową i ukryte poszerzono o pojedyncze neurony bias b_1, b_2 i b_3 o wartościach 1. W sumie sieć składa się z 13376 wag normalnych i 58 wag bias. Jako funkcję straty zastosowano categorical crossentropy.



Rysunek 5.3. Schemat modelu sieci neuronowej zastosowanego w eksperymentach do danych rozpoznawania cyfr MNIST.

W ramach testu kontrolnego zbadano trafność predykcji sieci na całym zbiorze testowym o wielkości 10000 próbek. Sieć nauczono za pomocą biblioteki Keras na pełnym zbiorze treningowym o wielkości 60000 próbek. Trening wykonano na optymalizatorach SGD oraz Adam. W obu przypadkach trwał on 10 epok. Dla SGD wykorzystano learningRate o wartości 0.1 oraz mini_batch o wielkości 50 - trafność wyniosła aż 95.85%. Test z wykorzystaniem optymalizatora Adam użył parametry learningRate o wartości 0.001 i mini_batch = 50 - trafność wyniosła niewiele więcej, 95.97%.

W porównaniu do poprzedniego badania, uzyskano o wiele lepszą dokładność nauczonej sieci. Wynikło to ze znacznie większej liczby próbek w zbiorze danych treningo-

wych oraz z wiele większego rozmiaru sieci, dlatego w celu otrzymania porównywalnych wyników zmniejszono liczbę próbek treningowych do 900 i wykonano testy ponownie. Wykorzystując optymalizator SGD o learningRate zwiększonym do 0.25 oraz mini_batch o rozmiarze 20 uzyskano 85.34% celności predykcji. Optymalizator Adam z learningRate = 0.01 i mini_batch = 50 uzyskał 85.22% trafności. Są to dobre wyniki biorąc pod uwagę zmniejszenie rozmiaru zbioru danych treningowych do 15% jego inicjalnej wielkości. Ponadto w przypadku tego testu istotne było zapewnienie dość równomiernego rozkładu danych treningowych, ponieważ sieć powinna być w stanie rozpoznać każdą z cyfr, a w przypadku niedoboru niektórych z nich nie nauczyłaby się poprawnie. Rozkład ilości rekordów kategoryzowanych jako poszczególne cyfry wykorzystanych w tym badaniu przedstawiono w tabeli 5.19. Podział ten był wystarczająco dobry do uzyskania zadowalających wyników.

Tabela 5.19. Rozkład cyfr w zadaniu klasyfikacji. Zbiór treningowy o 900 rekordach.

Cyfra	0	1	2	3	4	5	6	7	8	9
Ilość	87	106	93	86	95	81	82	98	80	92

Następnie wykonano test nauczenia 3 osobnych modeli dla podzielonego zestawu 900 rekordów na 3 zbiory po 300 rekordów. W tabeli 5.20 przedstawiono wyniki dla optymalizatora SGD, a w tabeli 5.21 wyniki dla optymalizatora Adam. SGD dla wszystkich trzech modeli uzyskał dokładność niewiele poniżej 80%. Adam w 2 przypadkach osiągnął trochę powyżej 80%, ale dla modelu 3 celność predykcji spadła do zaledwie 71.93%. Wyniki te są o od kilku do nawet kilkunastu % gorsze od testu na łącznym zbiorze danych. Ich pogorszenie było spodziewane, ze względu na niewielkie rozmiary zbiorów treningowych w porównaniu do dużego rozmiaru sieci oraz aż 10 klas warstwy wyjściowej.

Tabela 5.20. Wyniki uczenia na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20) (%)

Model 1	Model 2	Model 3
79.11	79.43	78.34

Tabela 5.21. Wyniki uczenia na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20) (%)

Model 1	Model 2	Model 3
80.28	81.76	71.93

W tabeli 5.22 przedstawiono rozkład rekordów poszczególnych klas w każdym z trzech zbiorów. W niektórych przypadkach cyfry były rozłożone nierównomiernie - na przykład zbiór 1 miał tylko 21 rekordów o etykiecie 8 i aż 39, czyli prawie dwa razy więcej, o ety-

kiecie 1. Taki rozkład w połączeniu z bardzo małą próbką każdej cyfry prawdopodobnie spowodował powyższe pogorszenie trafności predykcji.

Tabela 5.22. Rozkład cyfr w zadaniu klasyfikacji. 3 zbiory treningowe po 300 rekordów.

Cyfra	0	1	2	3	4	5	6	7	8	9
Zbiór 1	34	39	28	34	32	23	29	29	21	31
Zbiór 2	24	40	36	25	27	28	25	33	28	34
Zbiór 3	29	27	29	27	36	30	28	36	31	27

W celu poprawienia dokładności modeli, na tych samych zbiorach danych zastosowano metodę scalania wag. W tabelach 5.23 oraz 5.24 przedstawiono wyniki tej metody odpowiednio dla optymalizatora SGD i Adam. Proces podzielono na 4 rundy, przez co w każdej rundzie każdy z modeli uczono na 75 rekordach. W obu symulacjach, tak jak w testach na zbiorze danych z cenami domów, pierwsza runda została wykorzystana na kalibrację wag rozbieżnych modeli. Ostatecznie SGD osiągnął 82.19%, a Adam 83.11%, więc w obu przypadkach trafność predykcji wzrosła w porównaniu do osobnego uczenia modeli.

Tabela 5.23. Wyniki symulacji komunikacji synchronicznej na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	56.98	62.92	59.98	20.14
2	68.67	71.85	61.04	74.58
3	76.58	73.96	78.48	80.90
4	79.86	80.36	75.52	82.19

Tabela 5.24. Wyniki symulacji komunikacji synchronicznej na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	60.16	67.64	62.96	30.96
2	71.89	68.77	64.12	74.74
3	75.70	72.42	73.52	81.35
4	76.14	78.92	73.56	83.11

Następnie przeprowadzono test na większej ilości rund - wyniki 6 rund przedstawiono w tabelach 5.25 oraz 5.26. Tym razem dokładność modeli pogorszyła się, dla Adam o 1%, a dla SGD o aż 4%. Dalsze zwiększenie liczby rund również przyniosło wyniki gorsze od testu na 4 rund - dla 10 rund końcowy model optymalizatora Adam osiągnął 80.68%, a SGD 78.99%. Oznacza to, że w przypadku zbioru rozłożonego na 10 klas ważny jest balans pomiędzy rozmiarem zbioru treningowego w danej rundzie a częstotliwością scalania modeli.

Tabela 5.25. Wyniki symulacji komunikacji synchronicznej na zestawie cyfr MNIST, 6 rund, optymalizator SGD (learningRate = 0.25, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	55.05	60.40	41.09	10.30
2	58.92	63.35	60.41	69.10
3	74.23	73.11	65.60	76.47
4	74.88	75.76	74.21	80.65
5	77.34	73.15	75.49	78.96
6	76.60	75.17	75.63	78.15

Tabela 5.26. Wyniki symulacji komunikacji synchronicznej na zestawie cyfr MNIST, 6 rund, optymalizator Adam (learningRate = 0.01, batch_size = 20) (%)

Runda	Model 1	Model 2	Model 3	Model scalony
1	59.44	61.49	50.26	26.06
2	61.40	63.62	63.80	69.27
3	69.89	70.14	60.28	75.22
4	70.19	67.01	69.83	76.51
5	75.74	61.34	71.55	78.28
6	75.57	78.80	66.73	82.02

Następnie wykonano testy komunikacji asynchronicznej, czyli ze średnią ważoną na podstawie złotego podziału. Tabele 5.27 i 5.28 przedstawiają wyniki tego testu. Optymalizator SGD dla wszystkich modeli osiągnął wyniki porównywalne do symulacji komunikacji synchronicznej. Adam osiągnął dokładność o zaledwie 0.48 - 1.15% gorszą od tego samego eksperymentu, ale wciąż poprawił celność predykcji modeli w porównaniu do testu bez wymiany wag (tabela 5.21), przy czym dla modelu 3 o aż 9.08%.

Tabela 5.27. Wyniki symulacji komunikacji asynchronicznej na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	56.98	62.92	59.98	20.63	22.26	20.16
2	69.69	71.69	61.81	69.93	72.13	70.22
3	76.30	73.99	77.63	80.44	80.32	80.43
4	78.58	80.33	74.41	81.96	82.63	82.29

5.1.4. Klasyfikacja cyfr MNIST - uśrednianie gradientów

Zbadano również metodę scalania gradientów na zbiorze danych MNIST. W tabeli 5.29 przedstawiono wyniki przebadania tej metody z wykorzystaniem optymalizatora SGD i parametrów takich samych jak w testach metody scalania wag. Niestety w przypadku tego zbioru danych uzyskano wyniki gorsze od kontrolnych - wynikowe modele osiągnęły trafność mniejszą o odpowiednio 2.59%, 4.04% i 3.87%.

Tabela 5.28. Wyniki symulacji komunikacji asynchronicznej na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	60.16	67.64	62.96	25.52	28.54	34.66
2	70.35	67.57	68.40	72.03	73.52	72.81
3	74.17	70.92	71.14	78.77	79.05	78.84
4	77.28	76.96	71.62	80.82	81.68	81.01

Tabela 5.29. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	56.98	62.92	59.98	42.19	51.47	47.68
2	70.84	70.60	68.74	70.67	69.76	69.29
3	74.63	67.19	73.51	74.27	66.35	72.67
4	77.71	75.37	73.61	76.52	75.39	74.47

W tabeli 5.30 przedstawiono wyniki tego samego badania dla optymalizatora Adam przy parametrach stałych względem poprzednich eksperymentów. Tak jak SGD, Adam uzyskał wyniki gorsze o 6.16%, 6.23% i 2.3% w porównaniu do testu kontrolnego oraz nie zdołał nawet poprawić trafności predykcji końcowego modelu 3, w przeciwieństwie do metody scalania wag.

Tabela 5.30. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	60.16	67.64	62.96	52.63	58.78	58.98
2	72.12	69.29	65.64	71.97	67.69	65.57
3	72.14	70.47	71.90	71.84	69.03	69.80
4	74.18	76.51	69.89	74.12	75.53	69.63

Następnie zastosowano parametr odcięcia gradientów gradThreshold o wartości 0.01 - wyniki testu dla SGD przedstawiono w tabeli 5.31. W porównaniu do poprzedniego testu, pomimo ucięcia gradientów o małych wartościach, dla modelu 1 uzyskano precyzję wyższą o 0.97%. Za to modele 2 i 3 straciły odpowiednio 1.35% i 3.52%. Mimo strat tych modeli, podczas całego procesu uczenia odcięto aż 23.9% wszystkich gradientów, co przy całkowitej liczbie wag równej 13434 jest na prawdę dużą liczbą.

Taki sam test przeprowadzono dla optymalizatora Adam - wyniki przedstawiono w tabeli 5.32. Przy odcięciu 17.39% wszystkich gradientów uzyskano precyzje lepsze od testu bez parametru gradThreshold (tabela 5.30) - dla modelu 2 o 2.05%.

Tabela 5.31. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20, gradThreshold = 0.01) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	56.98	62.92	59.98	42.37	51.75	47.94
2	71.10	70.44	68.98	70.66	69.83	69.31
3	74.30	70.68	73.86	73.91	68.22	72.79
4	77.62	74.48	71.44	77.49	74.07	70.95

Tabela 5.32. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20, gradThreshold = 0.01) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	60.16	67.64	62.96	52.66	58.82	58.96
2	71.98	69.40	65.48	71.69	67.71	65.40
3	72.77	71.88	71.47	72.67	69.10	70.14
4	75.14	79.16	70.28	74.97	77.58	69.78

Kolejnie zbadano wpływ zwiększenia wartości parametru gradThreshold do 0.1. W tabeli 5.33 przedstawiono wyniki dla optymalizatora SGD. Precyzja wynikowych modeli spadła o aż 3.84%, 5.44% i 3.18% w porównaniu do poprzedniego testu, z parametrem gradThreshold równym 0.01 (tabela 5.31). Wyniki te odbiegły od testu kontrolnego o 5.46%, 10.8% i 10.57% (tabela 5.20). Wynikło to z odcięcia aż 54.65% gradientów.

Tabela 5.33. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator SGD (learningRate = 0.25, batch_size = 20, gradThreshold = 0.1) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	56.98	62.92	59.98	50.91	60.89	57.47
2	71.10	69.85	66.03	71.19	69.87	66.00
3	71.78	72.33	72.30	71.95	71.41	72.65
4	74.48	68.78	68.94	73.65	68.63	67.77

Analogicznie przetestowano optymalizator Adam - tabela 5.34. Trafność predykcji wynikowych modeli znów spadła w porównaniu do poprzedniego testu, co było spodziewane. Utrata precyzji modeli 1 i 3 nie była tak drastyczna jak w przypadku optymalizatora SGD - spadła ona o kolejne 1.84% i 1.73%. Model 2 pogorszył się o 3.89%. Jednak tak jak SGD, odcięto ponad połowę wszystkich gradientów, aż 51.5%. Jest to cena za trafność predykcji gorszą o średnio 1.2% od wyników testu bez zastosowania parametru odcięcia (tabela 5.30).

Przy bardzo dużym rozmiarze sieci i dużym obciążeniu kanału komunikacyjnego metoda wykorzystująca parametr odcięcia gradientów może być bardzo opłacalna. Nie

Tabela 5.34. Wyniki symulacji wymiany gradientów na zestawie cyfr MNIST, optymalizator Adam (learningRate = 0.01, batch_size = 20, gradThreshold = 0.1) (%)

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	60.16	67.64	62.96	53.49	61.00	60.72
2	70.09	69.33	65.40	70.38	68.75	65.89
3	74.34	68.95	70.42	74.95	69.01	70.41
4	75.08	74.42	68.65	73.13	73.69	68.85

zmienia to jednak faktu, że wszystkie testy metody uśredniania gradientów uzyskały wyniki gorsze od kontrolnych (tabele 5.20 i 5.21), gdzie uczone modele nie wymieniły się żadnymi informacjami.

5.2. Rozproszone uczenie maszynowe na Arduino

W ramach badań na rzeczywistej sieci mikrokontrolerów, przeprowadzono testy na sieci 3 urządzeń Arduino Nano 33 BLE Sense. Do testów wykorzystano metodę optymalizacji SGD, jako że była to jedyna metoda zaimplementowana w wykorzystanej bibliotece uczenia maszynowego. Jako kanał komunikacyjny wykorzystano Bluetooth Low Energy - jest to jedyny dostępny środek wymiany danych bezpośrednio pomiędzy urządzeniami, który niestety ma swoje ograniczenia. Największym z nich jest sposób przesyłania danych, opisany w rozdziale 3., który pozwala przekazać w łatwy sposób do 244 bajtów naraz. Sieć neuronowa wykorzystana w badaniach ze zbiorem cyfr MNIST była zdecydowanie zbyt wielka, aby móc ją przekazywać z dużą częstotliwością, nawet z zastosowaniem liczb zmiennoprzecinkowych o mniejszej precyzji i metody uśredniania gradientów z obcinaniem wartości poniżej ustalonego parametru gradThreshold. Z tego powodu do badań na sieci mikrokontrolerów wykorzystano zbiór danych firmy Zillow, klasyfikujący ceny domów, o ustawieniach sieci takich samych jak w symulacjach.

Najpierw zbadano metodę scalania wag. Analogicznie do symulacji komunikacji asynchronicznej, urządzenia na początku każdej rundy otrzymały 75 rekordów, następnie wykonały na nich trening swojego modelu przez 10 epok i próbowały pobrać modele innych urządzeń jedno po drugim, uśredniając je ze swoim na podstawie średniej ważonej przy udanym połączeniu. Test wykonano z ustawieniem 4 rund oraz parametrem learningRate równym 0.1. W tabeli 5.35 przedstawiono trafności predykcji modeli tego testu - przykładowo model 1 i model scalony 1 odpowiadają modelowi na urządzeniu 1 kolejno po treningu i po próbach odczytania danych z dwóch pozostałych urządzeń. W przeciwieństwie do symulacji urządzenia nie były synchronizowane w każdej rundzie - oznacza to, że dane urządzenie mogło przejść przez proces scalania wag szybciej dzięki udanym połączeniom BLE, podczas gdy inne urządzenie nadal próbowało wykryć docelowy mikrokontroler przez skan. W przypadku urządzeń 2 i 3 uzyskano precyzje porównywalne zarówno do symulacji jak i badań kontrolnych. Dokładność modelu urzą-

dzenia 1 na zbiorze testowym wyniosła o kilka procent mniej od pozostałych. Podobnie jak w symulacjach, runda początkowa nie dała dobrych wyników, z wyjątkiem urządzenia 3, które po scaleniu wag uzyskało 78% dokładności precyzji. W pozostałych rundach większość precyzji utrzymała się na poziomie ponad 80% - wszystkie urządzenia szybko uzyskały podobne modele.

Tabela 5.35. Wyniki badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się wagami modeli (%).

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	50.75	50.75	59.75	50.75	59.25	78
2	82	82.5	84.75	77.25	82.5	84.5
3	82.5	80.5	85.5	82.5	80.5	85.5
4	78.25	84	83.25	78.25	84.25	83.5

Słabsza trafność predykcji urządzenia 1 w końcowej rundzie wynikła ze straty po ostatniej sesji treningu w połączeniu z nieudanymi odczytami z obu pozostałych urządzeń. W tabeli 5.36 przedstawiono liczby nieudanych prób odczytania danych poprzez Bluetooth Low Energy. Do powodów takich prób należą: niewykrycie docelowego urządzenia podczas skanu, nieudana próba połączenia i odczytanie błędnych danych (ciągu zer). W ciągu 4 rund przy 2 próbach na urządzenie, 11 z 24 prób nie powiodło się.

Tabela 5.36. Liczba nieudanych prób odczytania danych BLE podczas badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się wagami modeli. 0 oznacza 2 udane próby w danej rundzie, podczas gdy 2 oznacza 0 udanych prób.

Runda	Urządzenie 1	Urządzenie 2	Urządzenie 3
1	0	1	0
2	1	0	1
3	1	2	2
4	2	0	1

W teście zmierzono również czasy wykonania poszczególnych bloków kodu i przydzielono je do jednej z trzech kategorii: Serial - łączny czas poświęcony na pobieranie danych z serwera Python przez port serial, Trening - czas poświęcony na trenowanie modelu oraz BLE - całkowity czas komunikowania się z innymi urządzeniami w sieci. Zmierzono także całkowity czas od początku do końca wykonywania programu. W tabeli 5.37 przedstawiono te czasy dla powyższego testu. Pomimo wielu obliczeń związanych z uczeniem, czas poświęcony na te operacje był bardzo mały - dla każdego z urządzeń wyniósł około 1 sekundy. Kilkukrotnie dłuższe było pobieranie zestawu 75 rekordów w każdej rundzie - przez okres 4 rund pobranie wszystkich 300 rekordów zajęło każdemu z urządzeń ponad 6 sekund. Jednak najwięcej czasu wykorzystano na komunikację przez Bluetooth Low Energy. Wiązało się to głównie z operacją skanu w poszukiwaniu innych

urządzeń, która bardzo często zajmowała kilka sekund. Łącznie na wszystkie operacje odczytywania danych przez BLE urządzenia potrzebowały około 40 sekund.

Tabela 5.37. Czasy badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się wagami modeli (s).

Czas	Urządzenie 1	Urządzenie 2	Urządzenie 3
Serial	6.38	6.21	6.53
Trening	1.0	0.96	0.98
BLE	39.02	37.77	42.85
Całkowity	47.83	46.4	52.76

Niestety nawet przy powyższych wynikach, które można nazwać zadowalającymi, test należało uruchomić kilka razy. Wiązało się to z niestabilnym działaniem komunikacji BLE pomiędzy urządzeniami. Podczas niektórych uruchomień poszczególne urządzenia czasem nie były w stanie znaleźć pozostałych, przez co nie mogły odczytać ich danych w danej rundzie. Nawet przy udanym skanie i połączeniu, istniała możliwość, że dane urządzenie odczyta wystawione dane jako ciąg zer, co efektywnie również wynikło nieudanym połączeniem.

W kolejnym badaniu z wykorzystaniem urządzeń Arduino przetestowano metodę uśredniania gradientów. Metoda ta, tak jak w symulacjach, sprawdzała się lepiej dla 6 rund uczenia. Wyniki testu, przedstawione w tabeli 5.38, zapewniają trochę lepsze i bardziej regularne precyzje w porównaniu do testu z wykorzystaniem metody scalania wag. Wynikło to jednak również z faktu, że przy większej liczbie rund urządzenia miały więcej szans na pomyślne próby odczytania danych pozostałych urządzeń. Urządzenie 1 w czwartej rundzie uzyskało spadek precyzji podobny do powyższego testu, ale w tym przypadku nadrobiło tą stratę w kolejnych rundach.

Tabela 5.38. Wyniki badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się gradientami (%).

Runda	Model			Model Scalony		
	1	2	3	1	2	3
1	50.75	50.75	59.75	52.5	52.5	71.25
2	82.25	81.5	84.75	79.75	79.5	81
3	83.75	75.75	84.75	83.75	75.75	84.75
4	78.5	83.75	83	78.5	83.75	83
5	83.25	81	83.25	83.5	84.25	83.25
6	84.75	85.75	83.5	84.25	84.25	83.5

Niestety w tym badaniu komunikacja Bluetooth Low Energy bardziej zawiodła. W tabeli 5.39 przedstawiono liczby nieudanych odczytów podczas tego badania. Aż 25 z 36 (69.4%) prób odczytu danych zakończyło się niepowodzeniem. W rundzie 3 i 4 żadne z

urządzeń nie uzyskało danych z sieci, a urządzenie 3 odczytało dane innego urządzenia łącznie jedynie 2 razy (83.3% nieudanych prób) - w rundzie 1 i 2.

Tabela 5.39. Liczba nieudanych prób odczytania danych BLE podczas badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się gradientami. 0 oznacza 2 udane próby w danej rundzie, podczas gdy 2 oznacza 0 udanych prób.

Runda	Urządzenie 1	Urządzenie 2	Urządzenie 3
1	0	1	1
2	1	1	1
3	2	2	2
4	2	2	2
5	1	1	2
6	1	1	2

W tabeli 5.40 zaprezentowano czasy przebiegu badania z metodą uśredniania gradientów. Czasy treningu oraz pobierania danych treningowych z serwera są podobne do poprzedniego testu. Czas komunikacji BLE znacznie się wydłużył. Pierwszym, oczywistym powodem, jest wydłużenie całości procesu uczenia do 6 rund. Drugim powodem jest większa liczba nieudanych prób odczytu danych z innych urządzeń. Dodatkowo w poprzednim teście większość nieudanych odczytów wynikła z przerwania połączenia lub niepoprawnego odczytu danych, co jest stosunkowo szybkim procesem. W tym badaniu wiele nieudanych odczytów polegało na niewykrywaniu pozostałych urządzeń podczas skanu, co powodowało przerwanie skanu dopiero po upływie czasu określonego wartością *SCAN_TIMEOUT*, która w obu testach wynosiła 10000 (10 sekund).

Tabela 5.40. Czasy badania przeprowadzonego na zestawie cen domów na 3 urządzeniach Arduino wymieniających się gradientami (s).

Czas	Urządzenie 1	Urządzenie 2	Urządzenie 3
Serial	6.65	7.84	4.93
Trening	1.5	1.46	1.46
BLE	90.03	84.54	86.84
Całkowity	100.54	96.25	95.38

5.2.1. Wnioski

Obie zaimplementowane metody - scalania wag i uśredniania gradientów - uzyskały precyzje na poziomie zadowalającym, jednak ze względu na ilość nieudanych prób odczytów danych innych urządzeń, ciężko jest stwierdzić czy efekt ten został osiągnięty dzięki komunikacji pomiędzy urządzeniami, czy przez zwykłe douczanie modeli na lokalnych danych w kolejnych rundach.

Operując na urządzeniach małego formatu należy zwrócić szczególną uwagę na rozmiar programu oraz ilość pamięci zajmowanej przez zmienne globalne i lokalne alokowane podczas wykonywania programu. Wybrane urządzenie, Arduino Nano 33 BLE Sense,

posiada 256 kB pamięci RAM oraz 1 MB pamięci flash - nie jest to wiele w porównaniu do pamięci dostępnej na przeciętnych komputerach, ale w kategorii małych mikrokontrolerów oznacza to urządzenie o dość dużej pojemności. Mimo to należy uważać na zapełnienie pamięci, ponieważ rozmiar sieci neuronowej skaluje się bardzo szybko w proporcji do wzrostu liczby jej neuronów.

Skompilowany program, który zaimplementowano w ramach tej pracy, zajmuje 70376 z 262144 bajtów (26.8%) pamięci RAM oraz 321452 z 983040 bajtów (32.7%) pamięci flash urządzenia. Pozwala to na wykorzystanie dość dużych sieci neuronowych.

Sieć wykorzystana do uczenia maszynowego na zbiorze domów o układzie {10, 4, 4, 1} ma 60 wag standardowych oraz 3 neurony bias. Z racji oszczędnego przechowywania wag bias w bibliotece NeuralNetworks, każda z nich jest określona tylko jedną wartością float, czyli wszystkie 3 razem zajmują jedynie 12 bajtów, podczas gdy wagi standardowe zajmują 240. Łącznie wszystkie wagi sieci mieszczą się w 252 bajtach. Dzięki temu przechowywanie dodatkowych kopii sieci (np. do obliczenia gradientów) oraz przeprowadzenie badań na wybranym kontrolerze nie stanowiło problemu.

Z kolei sieć wykorzystana do klasyfikacji ręcznie napisanych cyfr MNIST ma układ {784, 16, 32, 10}, czyli w sumie składa się z 13376 wag standardowych i 3 wag bias. Oznacza to, że jedna kopia listy wag zajmuje 53,516 bajtów (52.26 kB) w wersji z liczbami o pełnej precyzji (float 4-bajtowy) oraz o połowę mniej, czyli 26,758 bajtów (26.13 kB) w wersji z liczbami o mniejszej precyzji (float 2-bajtowy). Są to rozmiary około 212-krotnie większe od poprzedniej sieci. Pomimo tego zarówno obiekt sieci jak i 3 kopie listy jej wag mieszczą się na urządzeniu i program jest w stanie wykonywać obliczenia bez problemu wyczerpania pamięci RAM. W tym przypadku jednak problemem okazała się komunikacja poprzez Bluetooth Low Energy. Przy charakterystykach o wielkości do 244 bajtów, przesłanie tak dużej sieci przy dość dużej liczbie nieudanych prób odczytu wymaga bardziej zaawansowanego algorytmu komunikacji pomiędzy urządzeniami oraz o wiele dłuższych, nieprzerwanych połączeń.

Innym, potencjalnie ważnym, aspektem wykorzystania takiego rozwiązania jest zużycie energii. Wiele mikrokontrolerów posiada małą baterię, która musi im wystarczyć na bardzo długi czas operacji. Wiąże się to z optymalizacjami programu i oszczędnym korzystaniem z energochłonnych komponentów. W przypadku zaimplementowanego rozwiązania elementem najbardziej ograniczającym długą żywotność baterii byłaby komunikacja Bluetooth Low Energy. W przeprowadzonych badaniach wykorzystano urządzenia Arduino stale podłączone do komputera przez kabel micro USB, ale w środowisku bez stałego dostępu do zasilania podejście z ciągłym rozgłaszaniem pakietu reklamującego oraz częstymi próbami połączeń mogłoby znacznie skrócić czas działania na baterii.

6. Podsumowanie

Celem niniejszej pracy było zbadanie możliwości sieci mikrokontrolerów w sferze rozproszonego uczenia maszynowego przeprowadzanego na urządzeniach komunikujących się ze sobą bezpośrednio, dążącego do zbieżnego modelu całej sieci na każdym z nich. Cel ten uzyskano poprzez analizę symulacji w środowisku Python z wykorzystaniem bibliotek TensorFlow i Keras oraz implementację rzeczywistej sieci mikrokontrolerów złożonej z trzech urządzeń Arduino Nano 33 BLE Sense komunikujących się ze sobą bezpośrednio. Omówiono również implementacje i zagadnienia z nimi związane.

W ramach symulacji przebadano dwie metody rozproszonego uczenia maszynowego. Pierwszą z nich było scalanie wag pomiędzy modelami z implementacją komunikacji synchronicznej, obecnej w rozwiązaniach korzystających z serwera pośredniczącego pomiędzy urządzeniami, gdzie wykorzystane zostają jednocześnie wagi wszystkich modeli w sieci i uzyskany model globalny jest przekazywany z powrotem do urządzeń. Następnie zbadano implementację tej metody, która imitowała komunikację bliższą asynchronicznej w sieci bez serwera, w której urządzenia wymieniają się danymi bezpośrednio pomiędzy sobą. W ostatniej części symulacji przetestowano również drugą metodę rozproszonego uczenia maszynowego, czyli metodę uśredniania gradientów w wersji standardowej oraz z wykorzystaniem liczb o mniejszej precyzji float16 i z obcinaniem mało znaczących przyrostów. Wszystkie symulacje wykonano na dwóch zbiorach danych - zbiorze klasyfikującym ceny domów oraz zbiorze cyfr pisanych ręcznie MNIST. Wyniki dla optymalizatorów SGD i Adam porównano w celu sprawdzenia, z którymi metodami lub zbiorami dany optymalizator radzi sobie lepiej.

Zaimplementowaną sieć mikrokontrolerów Arduino Nano 33 BLE Sense przebadano pod kątem możliwości realizacji takiego rozwiązania w sieci urządzeń niezależnych od serwera agregującego modele. Badania wykazały, że z częstotliwością połączeń obecną w testach problemem okazuje się komunikacja przez Bluetooth Low Energy, która w większości prób kończy się niepowodzeniem. Mimo to urządzenia poradziły sobie z główną kwestią, czyli uzyskaniem zadowalających w zakresie precyzji modeli.

Istnieje również możliwość rozwinięcia badań rozproszonego uczenia maszynowego na przedstawionym w niniejszej pracy układzie sieci mikrokontrolerów. Jako że głównym problemem w zaimplementowanym rozwiązaniu okazała się komunikacja pomiędzy urządzeniami, implementacje i badania przedstawionego rozwiązania z wykorzystaniem innych metod komunikacji byłyby wartościowe.

Kolejną kwestią wartą dalszej uwagi jest wykorzystanie niezrównoważonych zbiorów uczących na poszczególnych urządzeniach, do którego pożądany byłby zbiór danych przeznaczony do klasyfikacji rekordów do wielu klas, jak np. zbiór MNIST, który w tym przypadku nie został wykorzystany ze względu na rozmiary zastosowanej do niego sieci neuronowej.

Równie wartościowe byłoby przeprowadzenie badań na większej liczbie różnych me-

tod obliczania modeli w rundach uczenia. Inne sposoby scalania modeli lub przekazywania wartościowych danych mają szansę na uzyskanie szybszej zbieżności lub mniejsze obciążenie kanału komunikacyjnego pomiędzy urządzeniami.

Bibliografia

[1] P. Kairouz, H. B. McMahan, B. Avent i in., *Advances and Open Problems in Federated Learning*, 2019. DOI: 10.48550/ARXIV.1912.04977. adr.: https://arxiv.org/abs/1912.04977.

[2] P. Warden i D. Situnayake, *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[3] N. Llisterri Giménez, M. Monfort Grau, R. Pueyo Centelles i F. Freitag, "On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning", *Electronics*, t. 11, nr. 4, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11040573. adr.: https://www.mdpi.com/2079-9292/11/4/573.

[4] B. McMahan, E. Moore, D. Ramage, S. Hampson i B. A. y. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data", w *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, A. Singh i J. Zhu, red., ser. Proceedings of Machine Learning Research, t. 54, PMLR, kw. 2017, s. 1273–1282. adr.: https://proceedings.mlr.press/v54/mcmahan17a.html.

[5] Q. Yang, Y. Liu, T. Chen i Y. Tong, "Federated Machine Learning: Concept and Applications", *ACM Trans. Intell. Syst. Technol.*, t. 10, nr. 2, sty. 2019, ISSN: 2157-6904. DOI: 10.1145/3298981. adr.: https://doi.org/10.1145/3298981.

[6] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan i K. Gopalakrishnan, *PACT: Parameterized Clipping Activation for Quantized Neural Networks*, 2018. DOI: 10.48550/ARXIV.1805.06085. adr.: https://arxiv.org/abs/1805.06085.

[7] *Strona TensorFlow Lite*, Dostęp zdalny (04.09.2022): https://www.tensorflow.org/lite/microcontrollers, 2022.

[8] S. Disabato i M. Roveri, "Incremental On-Device Tiny Machine Learning", w *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, ser. AIChallengeIoT '20, Virtual Event, Japan: Association for Computing Machinery, 2020, s. 7–13, ISBN: 9781450381345. DOI: 10.1145/3417313.3429378. adr.: https://doi.org/10.1145/3417313.3429378.

[9] F. Yu, W. Zhang, Z. Qin i in., *Fed2: Feature-Aligned Federated Learning*, 2021. DOI: 10.48550/ARXIV.2111.14248. adr.: https://arxiv.org/abs/2111.14248.

[10] I. Tenison, S. A. Sreeramadas, V. Mugunthan, E. Oyallon, E. Belilovsky i I. Rish, *Gradient Masked Averaging for Federated Learning*, 2022. DOI: 10.48550/ARXIV.2201.11986. adr.: https://arxiv.org/abs/2201.11986.

[11] *Biblioteka NeuralNetworks*, Dostęp zdalny (04.09.2022): https://github.com/GiorgosXou/NeuralNetworks, 2021.

[12] Martín Abadi, Ashish Agarwal, Paul Barham i in., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. adr.: https://www.tensorflow.org/.

[13] F. Chollet i in., *Keras*, https://keras.io, 2015.

[14] Arduino, *Biblioteka ArduinoBLE*, Dostęp zdalny (04.09.2022): https://www.arduino.cc/reference/en/libraries/arduinoble, 2022.

[15] *Wtyczka PlatformIO*, Dostęp zdalny (04.09.2022): https://platformio.org, 2022.

[16] J. L. W. En, *How to build your first Neural Network to predict house prices with Keras*, Dostęp zdalny (04.09.2022): https://www.freecodecamp.org/news/how-to-build-your-first-neural-network-to-predict-house-prices-with-keras-f8db83049159/, 2019.

[17] Y. LeCun, C. Cortes i C. Burges, "MNIST handwritten digit database", *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, t. 2, 2010.

Spis rysunków

Spis tabel

Spis fragmentów kodu

Spis załączników

Załącznik 1. Funkcja updateCharacteristics

```
1  void updateCharacteristics() {
2    float biases[numLayers-1];
3    float weightValues[NUM_WEIGHTS];
4    unsigned int counter = 0;
5
6    for (int i=0; i < mlNet.numberOflayers; ++i) {
7      for (int j=0; j < layers[i+1]; ++j) {
8        for (int k=0; k < layers[i]; ++k) {
9          float wVal = mlNet.layers[i].weights[j][k];
10         weightValues[counter] = wVal;
11         ++counter;
12       }
13     }
14     memcpy(&biases[i], mlNet.layers[i].bias, sizeof(float));
15   }
16
17   int weightsSize = sizeof(weightValues);
18   int biasesSize = sizeof(biases);
19   uint8_t charBytes[weightsTotalSize];
20   uint8_t biasesBytes[biasesSize];
21
22   memcpy(charBytes, weightValues, weightsSize);
23   memcpy(biasesBytes, biases, biasesSize);
24
25   bleChar.writeValue(charBytes, weightsSize);
26   biasesChar.writeValue(biasesBytes, biasesSize);
27 }
```

Załącznik 2. Funkcja controlPeripheral

```
1  bool controlPeripheral(BLEDevice peripheral) {
2    println("- Connecting to peripheral device...");
3
4    if (peripheral.connect()) {
5      println("* Connected to peripheral device!");
6      println(" ");
7    } else {
8      println("* Connection to peripheral device failed!");
9      println(" ");
10     return false;
11   }
12
13   println("- Discovering peripheral device attributes...");
14   if (peripheral.discoverService(bleServiceUuid)) {
15     println("* Peripheral device attributes discovered!");
16     println(" ");
17   } else {
18     println("* Peripheral device attributes discovery failed!");
19     println(" ");
20     peripheral.disconnect();
21     return false;
22   }
23
24   BLECharacteristic recWeightsChar =
25                     peripheral.characteristic(bleCharUuid);
26   BLECharacteristic recBiasesChar =
27                     peripheral.characteristic(bleChar2Uuid);
28
29   if (peripheral.connected()) {
30     const int weightsSize = sizeof(weightValues);
31     uint8_t weightsBytes[weightsSize];
32     recWeightsChar.readValue(weightsBytes, weightsSize);
33
34     const int biasesSize = sizeof(recBiases);
35     uint8_t biasesBytes[biasesSize];
36     recBiasesChar.readValue(biasesBytes, biasesSize);
37     memcpy(recValues, weightsBytes, sizeof(recValues));
38     memcpy(recBiases, biasesBytes, biasesSize);
39
40     peripheral.disconnect();
41     return true;
42     }
43   }
44   return false;
45 }
```

Załącznik 3. Funkcja readModel

```
1  def readModel(model, s):
2      sep = '----------------------'
3      weights = model.get_weights()
4
5      for idxLayer in range(0, NUM_LAYERS-1):
6          idx = s.find(sep)
7          s = s[idx + len(sep) + 2:]
8          idx2 = s.find('|')
9          nnShape = list(map(int, s[:idx2].split(" ")))
10         s = s[idx2:]
11
12         sBias = 'bias:'
13         idxB = s.find(sBias)
14         idxB2 = s.find('\n')
15         biasS = s[idxB + len(sBias) : idxB2]
16         if (biasS == 'ovf'):
17             biasS = '1.0'
18         bias = np.float32(biasS)
19
20         for i in range(0, nnShape[1]):
21             weights[idxLayer*2+1][i] = bias
22
23         sW = 'W:'
24         nnWeights = np.empty([nnShape[1], nnShape[0]], \
25                              dtype=np.float32)
26         for i in range(0, nnShape[1]):
27             for j in range(0, nnShape[0]):
28                 idxW = s.find(sW)
29                 idxWend = s[idxW+3:].find(' ')
30                 sWeight = s[idxW+2:idxW+2+idxWend]
31                 if (sWeight[1:] == 'ov' or sWeight[1] == 'a'):
32                     sWeight = ' 0'
33                 weight = np.float32(sWeight[1:])
34                 if (sWeight[0] == '-'):
35                     weight = -weight
36                 nnWeights[i][j] = weight
37                 weights[idxLayer*2][j][i] = weight
38                 s = s[idxW+2+idxWend:]
39     model.set_weights(weights)
40     return
```

Załącznik 4. Funkcje get_grads i apply_grads

```
w1 = model1.get_weights()
w2 = model2.get_weights()
w3 = model3.get_weights()

def get_grads(prev, curr):
    num_nonzeros_before = 0
    num_nonzeros_after = 0

    grads = []
    for i in range(0, len(prev)):
        grads_layer = curr[i] - prev[i]
        grads.append(grads_layer)

        if ('GRAD_THRESHOLD' in globals()):
            num_nonzeros_before += np.count_nonzero(grads[i])
            grads[i] = np.where(
                    abs(grads[i]) < GRAD_THRESHOLD, 0, grads[i])
            num_nonzeros_after += np.count_nonzero(grads[i])

    num_zerod = num_nonzeros_before - num_nonzeros_after

    grads_16 = []
    for i in range(0, len(grads)):
        grads_16.append(grads[i].astype(dtype=np.float16))

    statistics = {
        'num_nonzeros_before': num_nonzeros_before,
        'num_nonzeros_after': num_nonzeros_after,
        'num_zerod': num_zerod,
    }
    return grads_16, statistics

def apply_grads(model, grads):
    weights = model.get_weights()
    for i in range(0, len(grads)):
        weights[i] = weights[i] + (grads[i] * GRAD_LR)
    model.set_weights(weights)
    return
```