AKADEMIA GÓRNICZO-HUTNICZA
im. St. Staszica w Krakowie
WEAIiE, Katedra Automatyki
Laboratorium Biocybernetyki

Przedmiot:	**Wieloprocesorowe Systemy Wizyjne.**			
			PR18wsw503	
Temat projektu:				
	Rozpoznawanie obiektów na obrazach w reprezentacji zdarzeniowej			
Wykonali:	Marcin	Kowalczyk	kowalczyk@agh.edu.pl	
	Jadwiga	Piechota	jadwiga.piechota@onet.pl	
	magisterski	rok V	Automatyka i Robotyka	
Rok akademicki	2017/18	zimowy		
Prowadzący	dr inż.	Mirosław	Jabłoński	
wersja 0.0 Kraków, styczeń, 2018.				

Streszczenie

Celem projektu było zaprojektowanie i implementacja sieci neuronowej do rozpoznawania obiektów na obrazach w reprezentacji zdarzeniowej. Skorzystano z bazy danych *MNIST-DVS*, która zawiera obrazy cyfr zapisane w tej reprezentacji. W celu zmniejszenia ilości danych potrzebnych do przetworzenia oraz poprawy jakości obrazów wykonano proste przetwarzanie wstępne. Dane zostały następnie przekonwertowane do postaci, która może posłużyć do uczenia sieci neuronowej. Na koniec sprawdzono skuteczność w zależności od wielkości sieci.

Spis treści

1 Wstęp

1.1 Cel projektu

Celem projektu jest implementacja rozwiązania pozwalającego na rozpoznawanie obiektów na obrazach w reprezentacji zdarzeniowej. Idea reprezentacji zdarzeniowej polega na zaznaczaniu pikseli, dla których nastąpiła pewna zmiana w obrazie wejściowym kamery. Współrzędne wraz z typem zmiany (wzrost lub spadek wartości piksela) tego piksela następnie są wysyłane do zewnętrznego urządzenia, które przetwarza te dane. Metoda ta pozwala na zmniejszenie ilości danych, które potrzebne są do przesłanie z kamery i przetworzenia w innym urządzeniu. Projekt zakłada, że zapisane dane z reprezentacji zdarzeniowej są następnie przetworzone przez sieć neuronową zaimplementowaną w programie *MATLAB*.

1.2 Proponowane rozwiązanie

Zaproponowane i sprawdzone rozwiązanie wymaga wstępnego przetworzenia zarejestrowanych danych. Polega ono na agregowaniu współrzędnych pikseli z 10us. Dane te następnie formatowane jako obraz o rozmiarze 40×40px oraz jako wektor, którego kolejne elementy są kolejnymi elementami stworzonego obrazu. Stworzone w ten sposób wektory są danymi wejściowymi sieci neuronowej. Dane wyjściowe są zapisywane w postaci wektora one-hot. Jest to wektor, który ma na jednym miejscu jedynkę, a na pozostałych zera. Tak przygotowane dane poddane są uczeniu z nauczycielem przez głęboką sieć neuronową, ponieważ rozważany problem jest problemem klasyfikacji. Wykorzystane narzędzia to funkcje do uczenia sieci neuronowej z wykorzystaniem autoenkodera zawarte w narzędziu programu *MATLAB - Neural Network Toolbox*. Użyte zostały dwa autoenkodery, które następnie wspólnie z warstwą wyjściową typu *softmax* tworzą całą głęboką sieć neuronową. Zbiór wejściowy podzielono na trzy części: dane do uczenia, walidacji i testowania. W kontekście alternatywnego rozwiązania okazało się prostsze w implementacji. Uzyskano zadowalające efekty, zatem zaproponowane rozwiązanie jest skuteczne i może być z powodzeniem wykorzystywane w procesie rozpoznawania cyfr.

1.3 Alternatywne rozwiązania

Alternatywne rozwiązanie zakładało implementację z użyciem impulsowych sieci neuronowych. Sieci te stanowią stosunkowo nowe podejście, a sposób działania bardziej zbliżony do rzeczywistych procesów zachodzących w mózgu. Lepiej też naśladują naturalne neurony. Bazują na sposobie przekazywania przez nie informacji, jaką jest impuls elektryczny. Jak zostało napisane w [1], sieci te lepiej nadają się do zadań dynamicznych. Podejście to wydaje się być idealnym rozwiązaniem do postawionego tu problemu. Naturalne neurony generują impulsy, które są przekazywane dalej [2]. Tutaj sieć reagowałaby na impulsy, czyli zmiany położenia obrazu wejściowego, tzw. zdarzenia. W dostępnej bazie każda kolejna dana generuje nowe zdarzenia, a więc impulsy, wynikające ze zmiany położenia badanego obiektu. Pomimo faktu, iż sieć ta idealnie nadałaby się do postawionego problemu, nie zdecydowano się na jej implementację. To stosunkowo nowe podejście, dlatego nie znaleziono gotowych bibliotek, które mogłyby ułatwić proces uczenia, ponieważ napisanie funkcji do uczenia tej sieci to temat obszerny i wykraczający poza możliwości zadania projektowego. Dodatkowo do symulacji tego typu sieci potrzebna jest duża moc obliczeniowa [1], co wiąże się z koniecznością dostępu do komputera spełniającego pewne wymagania sprzętowe.

2 Analiza złożoności i estymacja zapotrzebowania na zasoby

Zadanie projektowe zostało podzielone na dwie części. Pierwsza z nich dotyczyła odpowiedniego przygotowania danych, czyli wybrania odpowiednich zdarzeń z bazy *MNIST-DVS* i dostosowania ich do aktualnych potrzeb. Dane te zostały następnie użyte podczas właściwego działania algorytmu, czyli uczenia i testowania sieci neuronowej.

2.1 Przygotowanie danych

Cała baza [3] zawierała bardzo dużo informacji. Nie było konieczności użycia tak dużej liczby wartości, dlatego zdecydowano się na wybranie około 10%. Liczba ta okazała się wystarczająca do zobrazowania wyników istotnych z punktu widzenia założonych celów. W celu znalezienia zbioru uczącego, napisano skrypt w programie *MATLAB 2017b*. Ta część zadania nie wymagała specjalnych zasobów obliczeniowych. Podstawowe wymagania systemowo - sprzętowe, potrzebne do poprawnego działania programu *MATLAB 2017b*, znalezione w [4], są następujące:

- system operacyjny: Windows 7 SP1 / Windows 8.1 / Windows 10
- procesor: Intel lub AMD x86-64 (rekomendowane wsparcie dla instrukcji AVX2)
- dysk: 4 - 6 Gb dla instalacji, 2 Gb wolnego miejsca dla prawidłowego działania programu
- RAM: co najmniej 2 Gb
- grafika: nie jest wymagana żadna konkretna karta graficzna, jednak polecana to taka, która ma wsparcie dla OpenGL 3.3 z 1 Gb pamięci GPU.

Zasoby pamięciowe, jakie dodatkowo musi posiadać komputer PC, na którym działa program, muszą pomieścić oryginalną bazę, która zajmuje około 9 Gb miejsca. Jednak już po przetworzeniu jej przez *MATLAB* zajmuje dużo mniej miejsca - to około 200 Mb. Aplikacja musi poradzić sobie z dużą ilością danych - na bazę składa się około 100 000 plików, przedstawiających zdarzenia przypadające na pewne przesunięcie danej liczby. Pliki te są zapisane w formacie .aedat, stworzonym przez twórców [5]. Autorzy stworzyli specjalny skrypt, który pozwala na zamianę formatu .aedat na format, który może być odczytany i dalej przetwarzany przez *MATLAB* (format .mat). Wymaga to wykonaniu dodatkowego szeregu instrukcji przy każdorazowym przetwarzaniu pliku z bazy, a wyniki tu generowane są przedstawione w formacie double. To wszystko powoduje dość długi czas trwania obliczeń, dlatego im lepsze parametry ma komputer PC, na którym wykonywane są obliczenia, tym wyniki uzyskiwane są sprawniej. Po wykonaniu obliczeń, uzyskuje się dwie macierze, wykorzystywane później do uczenia sieci. W obu wartościami są liczby boolean. Ponieważ nie ma tutaj żadnych zależności z pozostałymi modułami (baza została przygotowana wcześniej i nie koliduje z procesem uczącym), nie ma znaczenia na jakim komputerze będą wykonywane obliczenia. Możliwości sprzętowe powodują jednak, że efektywność pracy rośnie.

2.2 Uczenie sieci neuronowej

Właściwa część algorytmu, czyli uczenie i testowanie sieci neuronowej została również wykonana w programie *MATLAB 2017b*. Wymagania sprzętowo - programowe są tu

więc identyczne co te, wymienione w sekcji 2.1. Jednak tutaj zastosowano dwa różne podejścia do tematu.

Pierwsze podejście zakładało użycie gotowego GUI programu *MATLAB - Neural Network Toolbox*, stworzonego do uczenia i testowania sieci neuronowej. Jedynym wymaganiem był tu dostęp do tego narzędzia z poziomu programu *MATLAB*.

Drugie podejście polegało na stworzeniu sieci neuronowej za pomocą funkcji programu *MATLAB*. Dodatkowo, w celu przyspieszenia obliczeń, wykorzystano tutaj GPU (z ang. *graphics processing unit*). Aby taki algorytm mógł działać, niezbędna jest tutaj karta graficzna NVIDIA wyposażona w jednostkę obliczeniową, obsługująca architekturę obliczeniową CUDA (z ang. *Compute Unified Device Architecture*). Takie podejście jest uzasadnione, ponieważ uczenie sieci neuronowych to proces równoległy. Zatem zwiększenie zrównoleglenia obliczeń poprzez użycie dodatkowego procesora znacznie usprawni przebieg działania i spowoduje, że testowanie, polegające na wielokrotnym uczeniu sieci z różnymi parametrami będzie szybsze, a cały proces bardziej efektywny. Wydajność w tym przypadku wzrasta w znacznym stopniu.

W obu podejściach jako wektory wejściowe podane są macierze:

- danych - 94 490 $\times$ 1600
- wyjść - 94 490 $\times$ 10

W obu przypadkach w pierwszym podejściu były to wartości logiczne, w drugim zostały zapisane jako typ *double*.

3 Koncepcja proponowanego rozwiązania

Pierwsza część pracy polegała na stworzeniu bazy danych, na których następnie można było uczyć sieć neuronową. Jak zostało wspomniane w sekcji 2, około 10% danych dostarczonych przez autorów bazy *MNIST-DVS* w źródle [5] zostało wykorzystanych w niniejszej pracy. Ponieważ rozważano dwie koncepcje, dane zostały przygotowane do realizacji obu w podobny sposób, jednak na końcu zostały zapisane w innej formie, co zostanie omówione w późniejszej części rozdziału.

Dane zostały wybrane w sposób losowy, odrzucono tylko początkowe wartości, które mogłyby być nieco zaszumione i wprowadzać niepotrzebne rozbieżności. Mając już tak przygotowany zestaw danych, zauważono pewną powtarzalność w przedstawianiu zdarzeń. Otóż zdarzenia przedstawiające zmianę położenia liczby znajdują się w podobnym miejscu (jak już zostało wspomniane, nie cały zestaw danych, czyli zbiór zdarzeń został wzięty pod uwagę). Dzięki temu można było znacząco zmniejszyć wymiary danych wejściowych do rozmiaru 40×40. Taki zabieg przyspieszył i tak już zaawansowane obliczenia. W dalszej kolejności stworzono tablicę wartości boolean o rozmiarze 40×40 i przepisano tam wartość 1 w miejscach, gdzie wystąpiły zdarzenia. W ten sposób powstało 94 490 takich tablic (po około 10 000 na jedną cyfrę). Braki wynikają z błędów twórców bazy *MNIST-DVS*, ale zostały wyeliminowane w procesie działania zaprezentowanego algorytmu. Ponieważ uczenie sieci neuronowej zastosowanej w omawianej metodzie następowało z nauczycielem, potrzebne było stworzenie macierzy wyjść. Macierz do tego utworzona miała tyle wierszy, ile tablic wejściowych, zaś kolumn tyle co cyfr, czyli 10. Wartość 1 w danej kolumnie informowała tu z jaką cyfrą mamy w tym przypadku do czynienia - numer kolumny wskazywał cyfrę pomniejszoną o jeden (pierwsza kolumna dla 0).

Uczenie sieci odbyło się równolegle na dwa sposoby. Pierwszy wymagał przedstawienia danych w postaci wierszowej. Stworzono więc macierz mającą tyle wierszy, ile danych wejściowych i tyle kolumn, ile wartości przypadało na każdą daną, czyli 1600. Drugi sposób pozwalał na stworzenie tablicy celek. Dane wejściowe były przedstawione w taki sposób, że każda celka była tablicą 40×40, czyli jedną daną. Etykiety stworzono bazując na macierzy wyjść. Każda celka przestawiała tu wektor o rozmiarze 10, w którym na odpowiednim miejscu, oznaczającym daną cyfrę, znajdowała się wartość 1. Synchronizację otrzymano poprzez numer celki w danych wejściowych odpowiadał numerowi celki w tablicy wyjściowej zawierającej etykiety.

Pierwsze podejście uczenia sieci zakładało użycie GUI programu *MATLAB - Neural Network Toolbox*, a dokładnie jego części odpowiedzialnej za rozpoznawanie wzorców i klasyfikację danych - *nprtool*. Po wprowadzeniu tu wartości wejściowych i etykiet, ustala się liczbę neuronów ukrytych i można już przejść do nauki sieci neuronowej. *MATLAB* sam ustala tutaj liczbę danych potrzebnych do uczenia, walidacji i testowania. Program używa tutaj sieci neuronowej z użyciem algorytmu wstecznej propagacji błędów (z ang. *backpropagation*). Algorytm uczenia kończy działanie po udanej sześciokrotnej walidacji. Wyniki testowania można zaobserwować na stworzonych przez GUI statystykach.

Drugie podejście zostało stworzone po to, aby móc zastosować głęboką sieć neuronową. Uczenie tej sieci odbywa się używając macierzy wejściowej i etykiet wyjściowych, czyli jest to uczenie z nauczycielem. Dodatkowo wprowadzone parametry pozwalają na zastosowanie równoległości obliczeń na wielu rdzeniach oraz użycie procesora graficznego GPU. Zastosowanie jest uzasadnione ze względu na dużą liczbę danych wejściowych, co zostało omówione w sekcji 2. To wszystko pozwala na przyspieszenie tego etapu uczenia. Uzasadnione jest tu użycie autoenkodera w celu dodania warstw ukrytych. Użyto

tutaj GPU dodając odpowiedni parametr w funkcji *trainAutoencoder*, ustalona została też maksymalna liczba epok. W celu stworzenia pełnej sieci, dodano warstwę wyjściową. Uwzględniając fakt, że jest to problem klasyfikacyjny, stworzono funkcję aktywacji typu *softmax*. Tak nauczoną sieć poddano testowaniu. Wyniki uzyskane nieco różnią się od wyników uzyskanych w podejściu wcześniejszym, co zostanie omówione w sekcji 4.

4 Symulacja i testowanie

4.1 Modelowanie i symulacja

Weryfikacja poprawności przedstawionej koncepcji rozwiązania nie mogła zostać przedstawiona w niniejszym projekcie w sposób symulacyjny. Powodem tego jest fakt, że program nie był implementowany na żadnej platformie sprzętowej, tylko w programie *MATLAB*. Jednak zastosowano tutaj pewien sposób, który pozwalał sprawdzić, czy w zbiorze uczącym znajduje się odpowiednia liczba danych oraz czy dobrano prawidłową liczbę neuronów ukrytych. Dla tych testów posłużono się wbudowanym narzędziem programu *MATLAB - Neural Network Toolbox*, a dokładnie jego narzędziem do klasyfikacji *nprtool*. Proste GUI w sposób intuicyjny pomaga przy uczeniu i testowaniu różnych zależności. Po nauczeniu jest możliwość podglądu, jak sieć zachowywała się w poszczególnych epokach uczenia oraz jak wyglądają wyniki na zbiorze uczącym, testowym i walidacyjnym. GUI również dobiera odpowiednie proporcje pomiędzy tymi zbiorami. Dane użyte do testowania przedstawione były w postaci macierzy, gdzie w wierszach znajdowały się kolejne dane wejściowe. Podobnie rzecz się miała z etykietami - również była to macierz, w której wierszach znajdowały się etykiety do odpowiadających wierszy w macierzy danych wejściowych.

Kryterium jakościowym, jaki wprowadzono podczas pracy nad projektem był stosunek procentowy poprawnie rozpoznanych cyfr do wszystkich danych wejściowych w zbiorze testowym, wybranym losowo spośród całej stworzonej bazy. Podział zbioru danych wejściowych na zbiór uczący, testowy i walidacyjny był domyślny i pozostawał w stosunku 14:3:3 (70%, 15%, 15% zbioru danych wejściowych). Zastosowano stałe wartości, aby uniknąć efektu przeuczenia i pozwolić na rzeczywistą ocenę poziomu nauczenia sieci.

4.2 Testowanie a weryfikacja

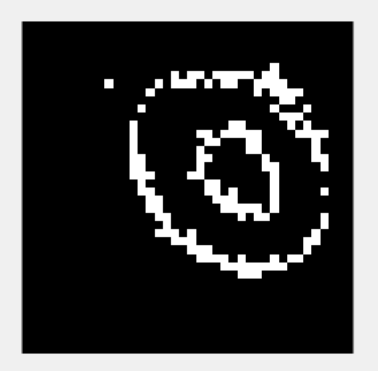
Docelowy algorytm, który został również napisany w programie *MATLAB 2017b*, uwzględniał już cały zbiór danych. Napisany został w sposób umożliwiający uruchomienie go równolegle na kilku rdzeniach procesora CPU i z użyciem procesora graficznego GPU. Optymalny czas uczenia uzyskuje się dysponując procesorem graficznym NVIDIA, wspierającym architekturę *CUDA*.

Wskaźnik jakości został tutaj wyznaczony identycznie jak w sekcji 4.1, podobnie było również podczas analizy ilościowej. Zbiór wejściowy został podzielony identycznie na zbiory: uczący, testujący i walidacyjny.

Eksperymenty praktyczne ograniczyły się do kilku tylko prób, z uwagi na dużą liczbę danych potrzebnych do uczenia, a co za tym idzie długi czas działania algorytmu. Optymalną ilość neuronów ukrytych uzyskano dzięki przeprowadzeniu symulacji, dlatego właściwy algorytm został od razu zaimplementowany z właściwymi parametrami. Zauważono pewną zależność - im więcej neuronów ukrytych dodawano, tym uczenie trwało dłużej. Dla około 100 000 danych wejściowych uczenie głębokiej sieci neuronowej zajmowało około 40 minut.

5 Rezultaty i wnioski

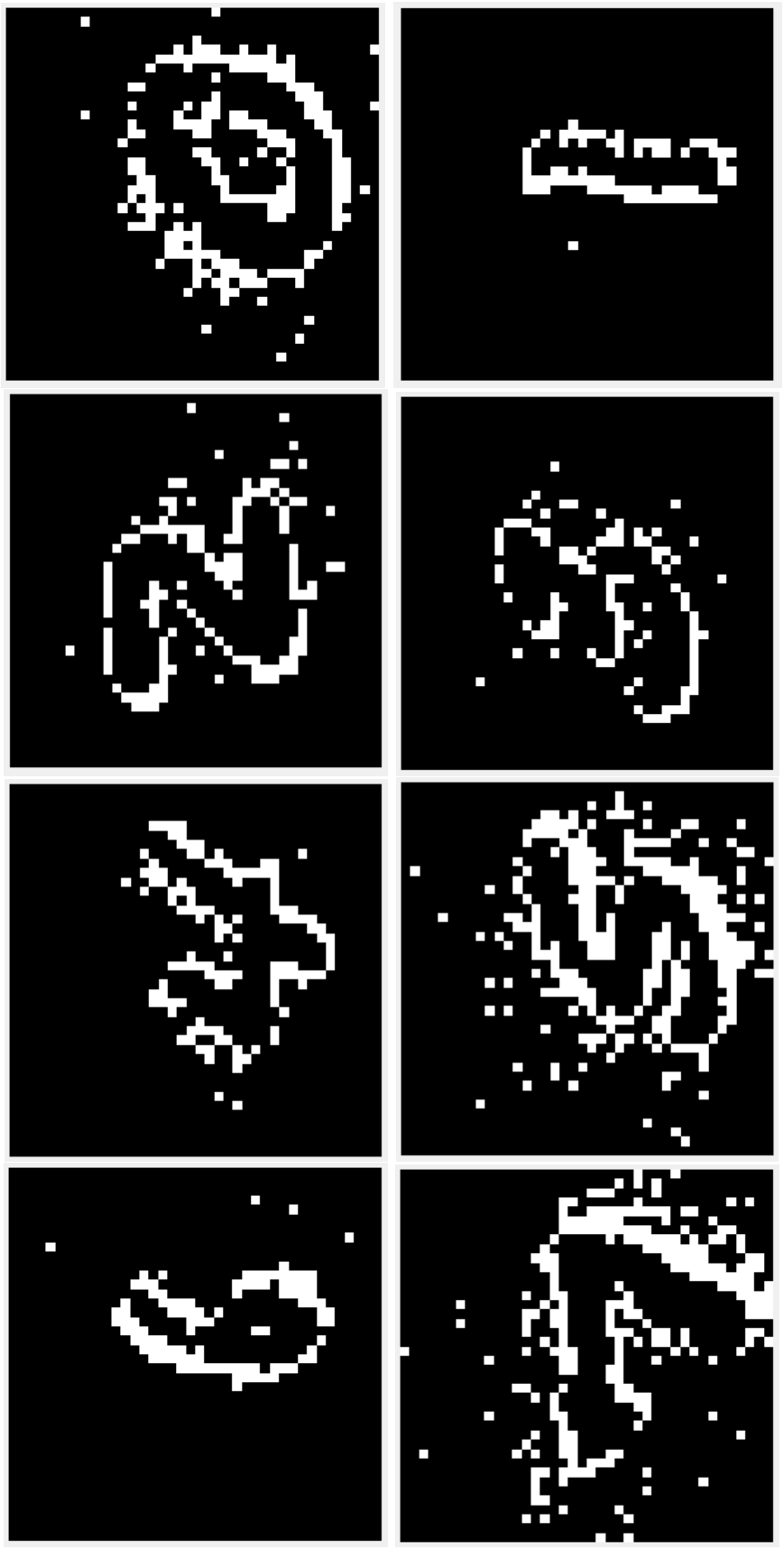
Rezultaty otrzymane podczas przetwarzania bazy *MNIST-DVS* pozwalały na sprawne i efektywne uczenie. Jak już zostało szczegółowo omówione w sekcji 3, dane wejściowe zostały przedstawione jako macierze o rozmiarze 40×40, w których wartość 1 oznacza, że nastąpiło zdarzenie w danym miejscu. Przykładowe dane zostały pokazane na rysunku 5.1 i 5.2.

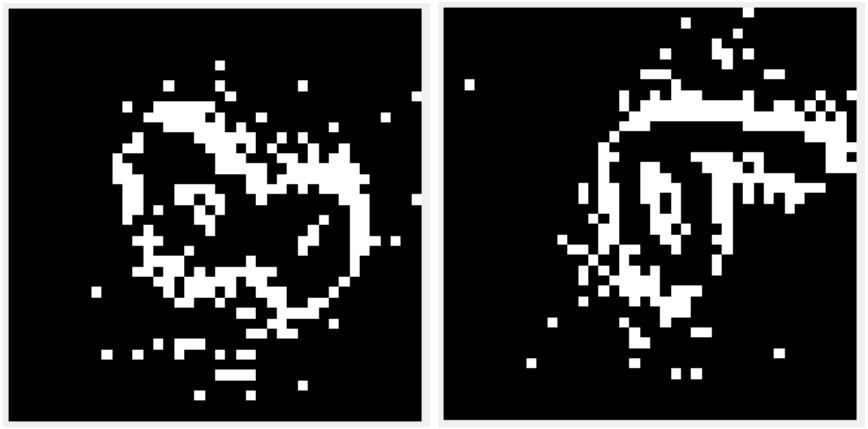


Rys. 5.1: Przykład niezaszumionych danych wejściowych przedstwiających zdarzenia reprezentujące cyfrę zero.

Rys. 5.2: Przykład zaszumionych danych wejściowych przedstwiających zdarzenia reprezentujące cyfrę osiem.

Zdarzenia reprezentujące cyfrę zero idealnie odwzorowują kontury obszaru zainteresowania. Jednak w przypadku cyfry osiem występują pewne szumy. Stanowią one duży problem dla człowieka, aby rozpoznać spośród licznych zdarzeń zdarzenia reprezentujące ruch konturów przedstawionej cyfry. Zaproponowana metoda jednak radzi sobie dobrze z tym problemem i prawidłowo rozpoznaje liczbę. Widać tu przewagę reprezentacji zdarzeniowej nad klasycznymi metodami, gdzie zaszumienie często powodowało, że sieć nie była poprawnie rozpoznać obiektów. Inne przykłady danych wejściowych zostały przedstawione na rysunku 5.3.





Rys. 5.3: Przykłady danych wejściowych - po jednym reprezentującym daną cyfrę.

5.1 GUI Matlaba - funkcja nprtool

Gotowe dane poddano symulacji używając narzędzia programu *MATLAB - nprtool*. Na zbiorze wejściowym testowano dano z użyciem różnej liczby neuronów ukrytych, w celu znalezienia wartości optymalnej. Wartość ta powoduje odpowiednio maksymalne wytrenowanie sieci, nie powodując przy tym jej przetrenowania. Dane przedstawiające jaki wpływ ma ilość neuronów ukrytych na skuteczność uczenia przedstawia tabela 1.

Tabela 1: Zależność liczby neuronów ukrytych od skuteczności uczenia sieci.

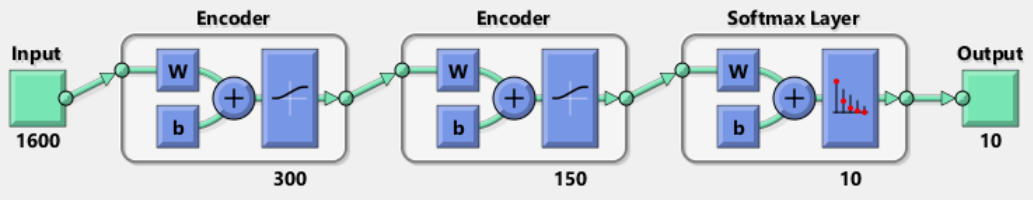
Ilość neuronów w warstwie ukrytej	Dane uczące [%]	Dane walidacyjne [%]	Dane testowe [%]	Cały zbiór wejściowy [%]
10	81.8	76.8	76.7	80.3
20	85.1	78.8	78.9	83.2
30	89.4	82.1	81.9	87.2
40	89	83.1	81.8	87.1
50	90.6	84.3	84.3	88.7
80	96.9	87.4	87.8	94.1
100	96	88.3	88.1	93.6
150	98.5	89.3	89.7	95.8
200	99	89.8	89.9	96.2
250	99.1	90.2	90.2	96.4
300	99.3	90.8	90.6	96.7
450	99.9	91.4	90.9	97.3

Widać tutaj dużą zależność pomiędzy ilościami neuronów ukrytych, a skutecznością rozpoznawania cyfr. Przy ilości neuronów 450 zdolność rozpoznawania dla zbioru uczącego

osiągnęła prawie 100%. Dla zbioru testowego, co jest właściwym wynikiem skuteczności działania algorytmu, wskaźnik jakości otrzymał wartość 90.9%. To wynik uśredniony dla wszystkich dziesięciu cyfr. Odchylenie standardowe danych wynosiło 2.7086%. Z uwagi na dużą liczbę danych, uzyskano dobrą skuteczność, pomimo iż część danych ze zbioru była mocno zaszumiona i algorytm nie jest w stanie ich rozpoznać.
Im więcej neuronów ukrytych, tym uczenie sieci trwa dłużej, nawet około 40 minut. Obciążenie procesora dochodzi tu do 80%. Jest to więc proces potrzebujący duże zasoby obliczeniowe. Prawdopodobnie dla większej liczby neuronów ukrytych uzyskano by większą skuteczność, jednak rozpoznawanie na poziomie 91% w reprezentacji zdarzeniowej było wystarczająco dobrym wynikiem.

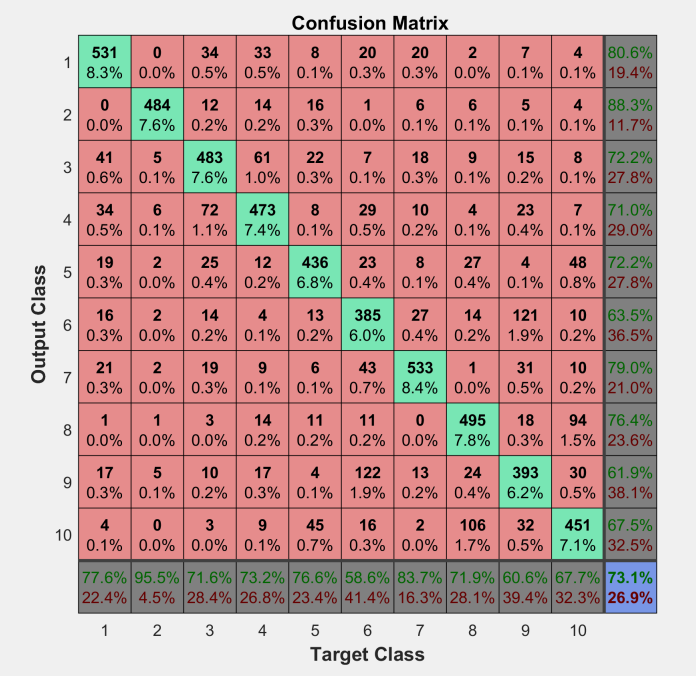
5.2 Skrypt do uczenia głębokiej sieci neuronowej

Sieć neuronowa na wejściu otrzymuje celki o rozmiarze 40×40, czyli łącznie 1600 elementów. Wynika z tego fakt, że liczba neuronów wejściowych jest również równa 1600. Podczas implementacji skryptu wybrano najbardziej optymalną ilość neuronów ukrytych. Zgodnie z wynikami uzyskanym i w sekcji 5.1 była to liczba 450. Zastosowano tu jednak trochę inne podejście - użyto dwa auto-enkodery - pierwszy posiadający 300, a drugi 150 neuronów. Na końcu zastosowano warstwę wyjściową typu *softmax*, która jako sygnały wejściowe używa wartości wyjść z drugiego auto-enkoder i pozwala na uzyskanie odpowiedniej liczby wyjść. Schemat całościowy sieci wraz z ilością neuronów w każdej warstwie przedstawiono na rysunku 5.4.



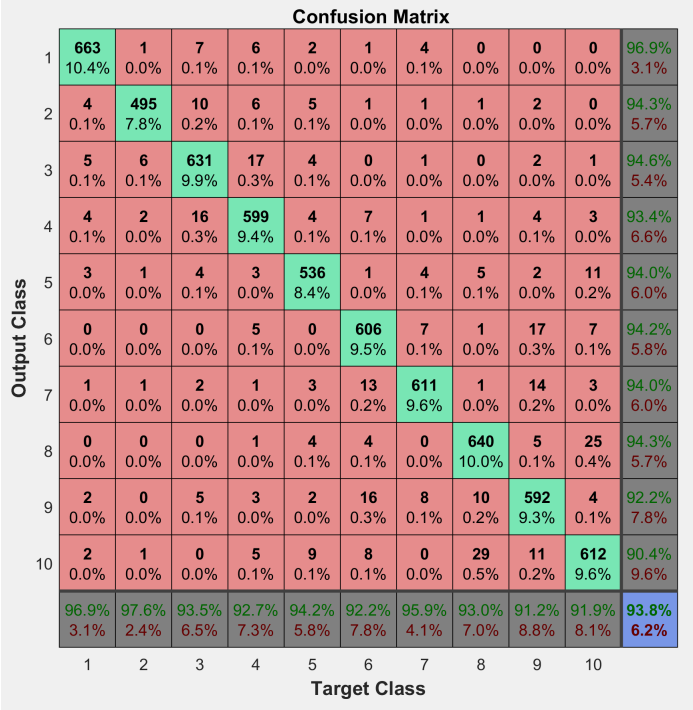
Rys. 5.4: Schemat zastosowanej głębokiej sieci neuronowej.

Dokładny opis funkcji użytych do stworzenia i nauczenia sieci znajduje się w sekcji B.1. Po wytrenowaniu sieci uzyskane wyniki przedstawiono na wykresie. Interesujące są te, które zostały otrzymane dla zbioru testowego, ponieważ to dla tych danych ustala się wskaźnik jakości algorytmu. Dokładne wyniki przedstawiające rozkład, które cyfry były najczęściej rozpoznawane lub mylone z innymi znajduje się na rysunku 5.5.



Rys. 5.5: Wynik uczenia głębokiej sieci neuronowej, posiadającej 450 neuronów ukrytych. Wyniki pokazane dla zbioru testowego stanowiącego 15% danych.

Wykres przedstawia skuteczność rozpoznawania przez sieć cyfr 0 - 9. W celu ułatwienia obliczeń cyfry zostały przeskalowane przez dodanie jedynki (1 - 10). Oś *Target Class* przedstawia rzeczywiste cyfry, zaś oś *Output Class* podaje informację, w jaki sposób algorytm rozpoznał te cyfry (do jakiej grupy je sklasyfikował). W każdym polu, u góry, zaznaczone pogrubioną czcionką znajduje się ilość danych sklasyfikowanych jak para (rzeczywista dana, wyjściowa dana). W tym samym polu na dole obliczony jest stosunek procentowy tych danych w całym zbiorze wejściowym określającym daną cyfrę. Pola oznaczone kolorem szarym przedstawiają zsumowane wyniki pól dla danej kolumny lub rzędu. Ostateczny wynik, uwzględniający zbiór cyfr całościowo, bez względu na rozróżnianie cyfr, znajduje się w prawym, dolnym rogu, oznaczony kolorem niebieskim.
Średnia wartość skuteczności, a więc przyjęty wskaźnik jakości otrzymał tu wartość 73.1%. To wynik uśredniony dla wszystkich dziesięciu cyfr. Odchylenie standardowe danych wynosiło 8.0810%. W przypadku tej metody nie uzyskano tak wysokiej skuteczności jak w poprzedniej, a więc do tego rodzaju danych lepiej sprawdziła się sieć typu *backpropagation*.
Zdecydowano się więc na dotrenowanie sieci poprzez użycie metody *backpropagation* dla wszystkich warstw. Uzyskano zadowalające efekty. Wyniki znajdują się na rysunku 5.6



Rys. 5.6: Wynik uczenia głębokiej sieci neuronowej, posiadającej 450 neuronów ukrytych, z użyciem metody *backpropagation*. Wyniki pokazane dla zbioru testowego stanowiącego 15% danych.

Średnia wartość skuteczności, a więc przyjęty wskaźnik jakości otrzymał tu wartość wysoką - 93.8%. To wynik uśredniony dla wszystkich dziesięciu cyfr. Odchylenie standardowe danych wynosiło 1.6753%. W przypadku tej metody uzyskano najwyższą skuteczność, a więc do tego rodzaju danych lepiej sprawdziła się sieć typu *backpropagation* z dwoma warstwami ukrytymi, mająca taką samą liczbę neuronów ukrytych co w poprzedniej metodzie mającej jedną warstwę.

Do wykonania tych obliczeń użyte były zarówno zasoby CPU jak i GPU, co pozwoliło zmniejszy czas obliczeń. Obciążenie procesora wynosiło około 70% maksymalnej wydajności. Uzyskano satysfakcjonujące wyniki, dlatego poprzestano testy na takiej liczbie neuronów.

5.3 Porównanie metod

Oba podejścia używają sieci typu *backpropagation* bazującej na metodzie gradientowej obliczania podczas aktualizacji wag. W *MATLAB* do realizacji tego celu służy finkcja *trainscg*. Różnica w obu podejściach polega na różnej liczbie warstw ukrytych - w pierwszym podejściu była jedna warstwa, w drugim dwie reprezentowane za pomocą stworzenia dwóch auto-enkoderów. Wyniki pokazują, że dwie warstwy sprawdziły się nieco lepiej w przypadku tego zadania. Ilość ukrytych neuronów pozostała taka sama w obu podejściach, jednak dzielenie procesu uczenia na dwie warstwy okazało się lepszym wyborem. Skuteczność na zbiorze testowym jest blisko 2% wyższa. Obie metody pokazują, że istnieje pewna analogia pomiędzy popełnianiem błędów w cyfrach podobnych do siebie pod pewnymi względami. Zauważono, że często algorytm rozpoznawał cyfrę '5' jako '6'

lub '8', a '8' jako '9' lub '5'. Ma to związek z faktem, że cyfry te mają dużo części wspólnych, a reprezentacja zdarzeniowa nie dostarcza pełnej informacji o krawędziach, ale tylko część tej informacji, czyli miejsce, gdzie nastąpiła pewna zmiana.

6 Podsumowanie

Zaproponowane rozwiązanie sprawdziło się przy założeniach postawionego problemu. Głębokie sieci neuronowe spowodowały dość dokładne rozpoznawanie, na poziomie 94%. Sieć ma duże możliwości, w miarę zwiększania liczby neuronów ukrytych lub liczby warstw zwiększała się skuteczność rozpoznawania cyfr na zbiorze testowym. Na zbiorze uczącym wartość ta była niespodziewanie wysoka (przy 450 neuronach dochodziła do 100%). Reprezentacja zdarzeniowa okazała się bardzo odporną na zakłócenia. W danych wejściowych istniała duża liczba zdarzeń zakłócających, tzn. nie należących do obiektu zainteresowań. Jednak mimo to sieć była w stanie poprawnie nauczyć się rozpoznawać te zaszumione dane wejściowe i poradziła sobie z problemem, z którym nawet człowiek miałby trudności. Gołym okiem często nie było widać z jaką cyfrą miało się do czynienia. Zastosowanie zrównoleglenia obliczeń oraz procesora graficznego GPU znacznie zwiększyło efektywność pracy i dało możliwość nauczenia sieci posiadającej w sumie 450 neuronów ukrytych. To wszystko dało zadowalający efekt końcowy, o dużej skuteczności rozpoznawania. Ponieważ zbiór uczący, walidacyjny i testowy był wybierany w sposób losowy, wyniki mogą delikatnie różnić się pomiędzy kolejnymi wywołaniami z tymi samymi parametrami.
Porównując obie metody okazuje się, że wyniki były nieco lepsze dla sieci składającej się z dwóch warstwy posiadających odpowiednio 300 i 150 neuronów ukrytych niż jednej o 450 neuronach. Przyjmuje się, że sieć z jedną warstwą ukrytą powinna nauczyć się rozwiązywania większości postawionych problemów i tak się stało, jednak z mniejszą skutecznością. Dodanie drugiej warstwy polepsza wskaźnik jakości. Obie sieci wykazywały jednak podobne błędy, myląc ze sobą te same cyfry. Ma to związek ze specyficzną postacią danych na wejściu w reprezentacji zdarzeniowej. Podsumowując, reprezentacja zdarzeniowa jest niecodziennym podejściem do problemu rozpoznawania cyfr. Jednak okazuje się, że można dzięki niej uzyskać całkiem zadowalające efekty, pomimo zaszumienia wynikającego z występowania losowych zdarzeń, nie związanych z poruszaniem się obiektu zainteresowania, czyli liczby. Niebagatelny wpływ ma na to częstotliwość pracy monitora czy kontrast ustawiony na ekranie. Nigdy baza nie będzie tutaj idealna, jednak widać, że pomimo tych niedoskonałości algorytm jest w stanie skutecznie określać z jaką cyfrą ma do czynienia.

Bibliografia

[1] Beata Joanna Grzyb. „Alternatywne modele neuronów i właściwości ich sieci". URL: https://www3.uji.es/~grzyb/downloads/master-thesis.pdf(term.wiz.2018-01-06). Praca magisterska. Uniwersytet Marii Curie-Skłodowskiej w Lublinie, 2007.

[2] Bartłomej Dzieńkowski. „Badanie możliwości wykorzystania impulsowej sieci neuronowej do symulacji żywych organizmów". URL: http://udkdev.net/portfolio/files/praca_dyplomowa.pdf(term.wiz.2018-01-06).. Praca dyplomowa. Politechnika Wrocławska, 2010.

[3] Teresa Serrano-Gotarredona i Bernabé Linares-Barranco. *Poker-DVS and MNIST-DVS. Their History, How They Were Made, and Other Details*. URL: https://www.frontiersin.org/articles/10.3389/fnins.2015.00481/full (term. wiz. 2017-12-19).

[4] *System Requirements - Release 2017b*. URL: https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/support/sysreq/files/SystemRequirements-Release2017a_SupportedCompilers.pdf/SystemRequirements-Release2017b_Windows.pdf (term. wiz. 2018-01-03).

[5] Teresa Serrano-Gotarredona i Bernabé Linares-Barranco. *Poker-DVS and MNIST-DVS. Their History, How They Were Made, and Other Details*. URL: https://www.frontiersin.org/articles/10.3389/fnins.2015.00481/full (term. wiz. 2017-12-19).

[6] *Sieci neuronowe - uczenie*. URL: http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad3/w3.htm (term. wiz. 2018-01-03).

[7] J. Piersa M. Czoków. *Algorytm propagacji wstecznej*. URL: http://www-users.mat.umk.pl/~piersaj/www/contents/teaching/wsn2010/wsn-lec06-bep.pdf (term. wiz. 2018-01-05).

[8] Adrian Horzyk. *Uczenie głębokie i głębokie sieci neuronowe*. URL: http://home.agh.edu.pl/~horzyk/lectures/miw/MIW-DL.pdf (term. wiz. 2018-01-05).

[9] Teresa Serrano-Gotarredona. *MNIST_DVS*. URL: http://www2.imse-cnm.csic.es/caviar/MNIST_DVS/ (term. wiz. 2017-12-19).

[10] Mark Beale Howard Demuth. *Neural Network Toolbox*. URL: http://cda.psych.uiuc.edu/matlab_pdf/nnet.pdf (term. wiz. 2018-01-05).

[11] *Construct Deep Network Using Autoencoders*. URL: https://www.mathworks.com/help/nnet/ug/construct-deep-network-using-autoencoders.html (term. wiz. 2018-01-05).

A DODATEK A: Szczegółowy opis zadania

A.1 Specyfikacja projektu

Tematem projektu było rozpoznawanie obiektów na obrazach w reprezentacji zdarzeniowej. Postawione zadanie wymagało znalezienia odpowiednich danych, które zostały przedstawione poprzez następujące po sobie zdarzenia. Podejście to nieco różni się od standardowego, trudno byłby wygenerować zbiór uczący w ramach realizacji tego zadania, bo to już jest temat na inny projekt. W literaturze poleconej przez prowadzącego znaleziono metodę, której wynikiem była baza *MNIST-DVS*. Baza ta została odnaleziona w [3] i wykorzystana do uczenia sieci neuronowej. Celem, jaki został postawiony, było nauczenie sieci neuronowej, bazując na zdarzeniowej reprezentacji danych, aby była w stanie rozpoznać 10 cyfr: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Wybór rodzaju sieci oraz platform sprzętowo - programowych prowadzący zostawił autorom projektu.

A.2 Szczegółowy opis realizowanych algorytmów przetwarzania danych

Sekcja ta będzie rozszerzeniem informacji zawartych już wcześniej w paragrafie 3. Zastosowane algorytmy zostaną tu omówione bardziej od strony matematycznej.

Przygotowanie danych Przetwarzanie bazy *MNIST-DVS* zostało napisane w programie *MATLAB*. Było to konieczne, ponieważ autorzy artykułu [5] zastosowali rozszerzenie .aedat. Dostarczyli dodatkowo skrypt *dat2mat*, który pozwalał na zamianę tych danych na pliki z rozszerzeniem .mat, które można już w łatwy sposób podglądnąć i przetworzyć. W tym przypadku zastosowano przetwarzanie sekwencyjne, tworząc skrypt, który zamieniał rozszerzenie i dostosowywał bazę do realizacji postawionych w projekcie celów.

Uczenie sieci Uczenie sieci przebiegało z użyciem wbudowanych funkcji programu *MATLAB*, co znacznie ułatwiło pracę. Dane były przetwarzanie współbieżnie, wykorzystując rdzenie procesora CPU oraz procesor graficzny GPU. Pierwszy sposób, uwzględniający GUI do uczenia sieci neuronowych, używał algorytmu wstecznej propagacji błędów i będzie omówiony w sekcji A.2. Drugi sposób polegał na wykorzystaniu wbudowanych funkcji do uczenia maszynowego i używał głębokich sieci neuronowych do klasyfikacji danych.

Algorytm wstecznej propagacji błędów To podstawowy algorytm uczenia nienadzorowanego wielowarstwowych sieci neuronowych. Zaletą tej sieci jest to, że wagi można tu wytrenować, znajdując ich optymalny zestaw [6]. Metoda umożliwia modyfikację wag w sieci o architekturze warstwowej, we wszystkich jej warstwach. Po ustaleniu topologii, początkowe wagi są inicjowane tu losowo. Przyjmują bardzo małe wartości. Następnie dla danego wektora uczącego oblicza się odpowiedź sieci, warstwa po warstwie, stosując algorytm spadku gradientowego [7]. Każdy neuron wyjściowy oblicza swój błąd, który następnie jest propagowany do wcześniejszych warstw. Następnie każdy neuron modyfikuje wagi na podstawie wartości błędu. Dodatkowo jest tu wprowadzona powtarzalność, czyli gdy wszystkie dane wejściowe zostaną już użyte, zmieniana jest ich kolejność i ponownie są wprowadzana do sieci. Proces trwa do momentu zatrzymania się średniego błędu kwadratowego. Jest to proces dość kosztowny obliczeniowo, zwłaszcza kiedy sieć jest rozbudowana.

Głębokie sieci neuronowe Uczenie głębokie sieci neuronowej następuje krok po kroku. Pozwala na stopniowe wyznaczenie wag dla poszczególnych warstw sieci w celu innej reprezentacji cech wspólnych. To poszczególne warstwy reprezentują tu cechy wspólne wzorców uczących i na tej podstawie tworzą reprezentacje bardziej skomplikowanych cech w kolejnych warstwach sieci głębokich. Jest to ulepszona metoda niż wielowarstwowe sieci neuronowe uczone algorytmem propagacji wstecznej, w której w warstwach oddalonych od wyjścia sieci sieć ma tendencję do dokonywania coraz mniejszych zmian [8]. Tutaj sieć jest rozbudowywana powoli o kolejne warstwy dopiero wtedy, gdy w poprzednich warstwach pojawiły się takie cechy. Stosuje się tu sieci typu *auto-encoder*, które używając aproksymacji identycznościowej wykorzystują warstwę ukrytą składającą się z mniejszej ilości neuronów niż ilość wejść lub wyjść z sieci. W przypadku głębokich sieci neuronowych trzeba uważać na to, aby sieć nie dopasowała się zbyt mocno do danych uczących, ponieważ na danych testowych może później niepoprawnie działać.

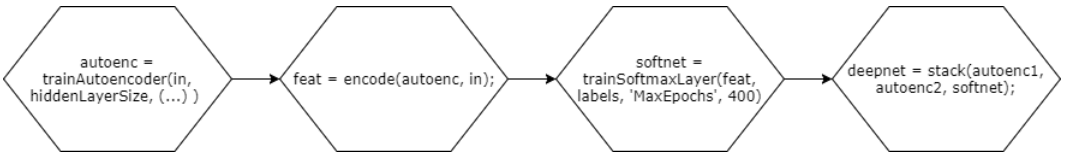
W przypadku realizacji przestawionego projektu wykorzystano głębokie sieci neuronowe do uczenia nadzorowanego, ponieważ rozważano problem klasyfikacji. Do realizacji głębokiego uczenia użyto tutaj auto-enkoderów, jako funkcja aktywacji posłużyła w warstwie wyjściowej funkcja typu *softmax*. Dokładny opis procedur znajduje się w sekcji B.

B DODATEK B: Dokumentacja techniczna

Projekt został napisany w programie *MATLAB 2017b*. Korzystano z gotowego GUI programu *MATLAB - Neural Network Toolbox* oraz innych funkcji z tego *toolbox*. Baza danych potrzebna do realizacji postawionego zadania została pobrana z źródła [9].

B.1 Dokumentacja oprogramowania

Wszystkie funkcje użyte na potrzeby zaimplementowania algorytmu zostały zaczerpnięte z gotowych rozwiązań. Opis i dokumentacja techniczna GUI *Neural Network Toolbox* znajduje się w źródle [10], zaś do funkcji użytych w procesie tworzenia głębokich sieci neuronowych w źródle [11].
Kolejność użytych funkcji przedstawiono na schemacie B.1. Dokładne omówienie kodu znajduje się poniżej.



Rys. B.1: Schemat blokowy użytych funkcji do głębokiego uczenia sieci neuronowej.

Funkcja *trainAutoencoder* tworzy ukryte warstwy o zadanej liczbie neuronów. Jako parametry przyjmuje dane wejściowe, ilość ukrytych neuronów oraz parametry określające dynamikę i właściwości wag. Dodatkowo można tu zdecydować, czy obliczenia mają być wykonywane również na procesorze GPU. Funkcja *encode* służy do ekstrakcji cech z ukrytych warstw, a jako parametry przyjmuje dane wejściowe oraz wynik działania auto-enkodera. Funkcja *trainSoftmaxLayer* tworzy funkcję aktywacji typu softmax, bazując na cechach zwróconych po wywołaniu funkcji *encode* oraz etykietach danych wejściowych. Określa się tutaj także maksymalną liczbę epok. Ostatnia funkcja *stack* układa wszystkie warstwy, łącząc stworzone auto-enkodery oraz warstwę wyjściową. Funkcja ta tworzy głęboką sieć neuronową, która może być już użyta do testowania.
Dokumentacja techniczna do każdej użytej funkcji znajduje się poniżej.

---

data = dat2mat(['path' '.aedat']);

@file Skrypt programu *MATLAB* napisany przez twórców[5] bazy *MNIST-DVS* do zamiany formatu danych .aedat na format .mat akceptowany i widziany przez *MATLAB*.
@param ['path' '.aedat'] Ścieżka do danych w formacie .aedat.
@return data Wynik działania skryptu. Przedstawienie danych podanych jako parametr w formacie .mat.

```
[trainInd,valInd,testInd] = dividerand(length(images), trainRatio, valRatio, testRatio);
```

@fn dividerand Funkcja dzieli określony ciąg liczb na trzy grupy używając losowych indeksów. W przypadku rozważanego algorytmu funkcja umożliwiła losowy podział zbioru wejściowego na zbiór: uczący, walidacyjny i testowy.
@param length(images) Liczba określająca wielkość zbioru, który ulega podziałowi. Tu: wielkość bazy danych wejściowych.
@param trainRatio Stosunek procentowy udziału danych w zbiorze treningowym. Default = 0.7.
@param valRatio Stosunek procentowy udziału danych w zbiorze walidacyjnym. Default = 0.15.
@param testRatio Stosunek procentowy udziału danych w zbiorze testowym. Default = 0.15.
@return trainInd Wektor indeksów danych treningowych.
@return valInd Wektor indeksów danych walidacyjnych.
@return testInd Wektor indeksów danych testowych.

```
autoenc1 = trainAutoencoder(images_train,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.004, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false, ...
    'useGPU',true);
```

@fn trainAutoencoder Funkcja trenuje auto-enkoder o zadanej liczbie neuronów ukrytych i kluczowych parametrach z punktu widzenia budowy sieci i użycia zasobów sprzętowych.
@param images_train Zbiór danych uczących w przypadku pierwszej warstwy ukrytej. W przypadku tworzenia kolejnych warstw (w rozważanym zadaniu użyto dwóch auto-enkoderów) jest to zbiór wyjściowy danych z poprzedniej warstwy ukrytej. Dane powinny być wyrażone w postaci macierzy lub tablicy celek. W każdym z przypadków dane powinny znajdować się w kolumnach bądź poszczególnych celkach o takim samym rozmiarze.
@param hiddenSize1 Liczba neuronów w warstwie ukrytej, jest to dodatnia wartość typu integer. Default = 10.
@param 'MaxEpochs' Maksymalna liczba epok treningowych, jest to dodatnia wartość typu integer. Default = 1000.
@param 'L2WeightRegularization' Współczynnik dla regulacji wag. Regulacja ta pozwala na uniknięcie zbytniego wzrostu wartości wag na rzecz wyjścia. Powoduje wyskalowanie tego parametru w funkcji kosztu. Jest to dodatnia wartość skalarna. Default = 0.001.
@param 'SparsityRegularization' Współczynnik kontrolujący wpływ parametru odpowiedzialnego za regulację wartości aktywacji neuronów w funkcji kosztu. Jest to dodatnia wartość skalarna. Default = 1.

@param 'SparsityProportion' Wskaźnik określający na ile przykładów danych treningowych ma reagować neuron. Im ta wartość jest mniejsza, tym neuron na mniej danych wejściowych będzie reagował wysokim wyjściem. Jest to dodatnia wartość skalarna z zakresu 0 - 1. Default = 0.05.
@param 'ScaleData' Parametr odpowiedzialny za skalowanie rozmiarem danych w auto-enkoderze automatycznie, co powoduje, że wejścia są replikowane na wyjścia. Przyjmuje wartość true lub false. Default = true.
@param 'useGPU' Parametr określający użycie GPU podczas trenowania. Przyjmuje wartość true lub false. Default = false.
@return autoenc1 Auto-enkoder stworzony z zadanymi parametrami, wytrenowany za pomocą danych wejściowych images_train.

---

plotWeights(autoenc1);

@fn plotWeights Funkcja wyświetlająca wagi dla neuronów ukrytych. Funkcja nie zwraca żadnych wartości, ale w wyniku jej działania na ekranie pojawia się wykres.
@param autoenc1 Auto-enkoder stworzony i wytrenowany za pomocą funkcji *trainAutoencoder*.

---

feat1 = encode(autoenc1, images_train);

@fn encode Funkcja tworzy i zwraca zakodowane dane, podane jako parametr, używając wytrenowanego wcześniej auto-enkodera. Dane te mogą być później wykorzystane jako wejście do kolejnych warstw ukrytych lub warstwy wyjściowej.
@param autoenc1 Auto-enkoder stworzony i wytrenowany za pomocą funkcji *trainAutoencoder*.
@param images_train Zbiór danych uczących w przypadku pierwszej warstwy ukrytej. W przypadku tworzenia kolejnych warstw (w rozważanym zadaniu użyto dwóch auto-enkoderów) jest to zbiór wyjściowy danych z poprzedniej warstwy ukrytej. Dane powinny być wyrażone w postaci macierzy lub tablicy celek. W każdym z przypadków dane powinny znajdować się w kolumnach bądź poszczególnych celkach o takim samym rozmiarze. Format tych danych zależy od tego, na jakim formacie danych trenowany był auto-enkoder.
@return feat1 Dane zakodowane za pomocą auto-enkodera i wyrażone w postaci macierzy. To cechy wygenerowane z danych wejściowych. Każda dana reprezentuje osobną kolumnę, ilość wierszy wskazuje na ilość neuronów ukrytych w auto-enkoderze.

---

labels_train_tempi = reshape(labels_traini, [size_x,size_y]);

@fn reshape Funkcja powoduje utworzenie nowej tablicy poprzez zmianę wymiarów tablicy już istniejącej.
@param labels_traini Tablica wzorcowa, która zostaje poddana zmianie rozmiaru.
@param [size_x,size_y] Nowy rozmiar tablicy.
@return labels_train_tempi Miejsce, w którym zostanie zapisana nowa tablica.

---

labels_train3 = cell2mat(labels_train_temp);

@fn cell2mat Funkcja konwertująca tablicę celek na standardową tablicę podstawowych typów.
@param labels_train_temp Tablica celek.
@return labels_train3 Tablica standardowa, utworzona z elementów znajdujących się poprzednio w celkach.

---

softnet = trainSoftmaxLayer(feat2, labels_train3, 'MaxEpochs', 400);

@fn trainSoftmaxLayer Funkcja stwarza warstwę wyjściową i klasyfikuje cechy z warstwy poprzedniej przy użyciu etykiet rzeczywistych.
@param feat2 Dane treningowe. Mogą to być zarówno dane wejściowe jak i cechy zwrócone przez poprzednią warstwę. W kolumnach znajdują się poszczególne próby, w wierszach kolejne cechy.
@param labels_train3 Rzeczywiste etykiety danych. Liczba wierszy specyfikuje liczbę klas, a więc wyjść z sieci, zaś każda kolumna specyfikuje kolejną daną. Trzeba zaznaczyć, że w danej kolumnie tylko jedna cyfra ma wartość 1, pozostałe mają wartość 0 i ta jedynka specyfikuje przynależność danej do tej konkretnej klasy.
@param 'MaxEpochs' Maksymalna liczba epok treningowych, jest to dodatnia wartość typu integer. Default = 1000.
@return

---

deepnet = stack(autoenc1, autoenc2, softnet);

@fn stack Funkcja układa w stos enkodery z kilku auto-enkoderów, tworząc pełną sieć neuronową.
@param autoenc1 Auto-enkoder pierwszy stworzony i wytrenowany za pomocą funkcji *trainAutoencoder*.
@param autoenc2 Auto-enkoder drugi stworzony i wytrenowany za pomocą funkcji *trainAutoencoder*.

@param softnet Warstwa wyjściowa stworzona za pomocą funkcji *trainSoftmaxLayer*.
@return deepnet Obiekt sieci neuronowej składający się z enkoderów elementów określonych jako parametry funkcji.

---

view(deepnet);

@fn view Funkcja pokazująca strukturę sieci podanej jako parametr z określeniem ilości neuronów i warstw. Funkcja nie zwraca żadnych wartości, ale w wyniku jej działania na ekranie pojawia się diagram.
@param deepnet Obiekt sieci neuronowej stworzony na przykład za pomocą funkcji *stack*.

---

y = deepnet(xTest);

@fn deepnet Wywołanie to pozwala na uzyskanie wyników działania sieci neuronowej na danych ze zbioru testowego. Używa tutaj stworzonego wcześniej obiektu sieci neuronowej deepnet.
@param xTest Zbiór danych wejściowych wyrażony jako tablica licz typu *double*. Liczba wierszy stanowi o rozmiarze tych danych, liczba kolumn wskazuje na ich liczbę.
@return y Wynik działania sieci neuronowej. Jest to tablica, której liczba kolumn informuje o liczbie danych wejściowych, a liczba wierszy wskazuje na liczbę neuronów w warstwie wyjściowej (i ich odpowiedzi dla danej o tym samym numerze kolumny).

---

plotconfusion(labels_test3, y);

@fn plotconfusion Funkcja pozwala na odczytanie macierzy błędu, czyli zbadanie związku pomiędzy właściwą klasyfikacją danych wejściowych a tą, zwróconą przez sieć neuronową. Funkcja nie zwraca żadnej wartości, ale w wyniku jej działania na ekranie pojawia się macierz opisująca analizę ilościową i jakościową testów.
@param labels_test3 Oryginalne etykiety danych poddanych testowaniu przez sieć.
@param y Etykiety wygenerowane przez sieć neuronową po przepuszczeniu przez nią danych testowych.

---

deepnet = train(deepnet, xTrain, labels_train3);

@fn train Funkcja trenuje sieć neuronową zgodnie z zadanymi parametrami, wykorzystując metodę *backpropagation*. Zwraca nową sieć, zmodyfikowaną sieć.
@param deepnet Obiekt sieci neuronowej stworzony na przykład za pomocą funkcji *stack*
@param xTrain Dane treningowe. Dane znajdują się w macierzy liczb double o liczbie wierszy wskazującej na wielkość jednej danej, w kolejnych kolumnach znajdują się kolejne dane testowe.
@param labels_train3 Oryginalne etykiety danych poddanych trenowaniu przez sieć. Etykiety znajdują się w poszczególnych wierszach, numer kolumny odpowiada danej, dla której została przyporządkowana etykieta.
@return deepnet Obiekt sieci neuronowej stworzony na przykład za pomocą funkcji *train*, używający metody *backpropagation* w procesie uczenia.

B.2 Procedura symulacji, testowania i weryfikacji

Do realizacji projektu nie używano żadnej zewnętrznej platformy sprzętowej. Aby uruchomić aplikację wystarczy komputer PC spełniający wymagania sprzętowo - programowe opisane w sekcji 2.1 z zainstalowanym programem *MATLAB 2017b*. Potrzebny jest także *Neural Network Toolbox*, który będzie wbudowany w przypadku pełnej instalacji programu. W celu przygotowania danych do uczenia można pobrać bazę ze źródła [3] i uruchomić skrypt *Data_selection* znajdujący się na nośniku CD. Można również wczytać przygotowane już dane w formacie .mat: data_matrix.mat oraz images_double.mat znajdujące się również na nośniku CD. W celu weryfikacji algorytmu należy skorzystać z *Neural Network Toolbox*, wybierając opcję *Pattern recognition and classification*. Jako wejście należy wczytać macierze input_matrix oraz output_matrix. Druga opcja zakłada skorzystanie z głębokiego uczenia z użyciem większej liczby warstw ukrytych. W tym celu trzeba uruchomić skrypt *GPU_Mnist* (płyta CD). Należy zaznaczyć, że proces trwa dosyć długo, a liczenie przebiega tu w sposób równoległy i z użyciem procesora graficznego GPU.

C DODATEK C: Spis zawartości dołączonego nośnika (płyta CD ROM)

Struktura folderów nośnika wygląda następująco:

- ◇ DOC - raport z projektu w formacie PDF
- ◇ SRC - zawiera skrypty źródłowe, Matlab2017b
  - dat2mat - stworzony przez autorów bazy [5], pozwala na zmianę rozszerzenie plików .aedat na format .mat
  - Data_selection - skrypt stworzony do przygotowania danych do uczenia sieci neuronowej
  - GPU_Mnist - algorytmy uczenia i testowania głębokiej sieci neuronowej użyciem równoległych obliczeń i procesora GPU
- ◇ TEST - znajdują się tu zarówno pliki do uczenia i testowania sieci neuronowej, jak i zapis z *workspace* po wytrenowaniu sieci: z udziałem GUI (wynik_450.mat) oraz auto-enkoderów (gpu_wynik_tune.mat).

D DODATEK D: Historia zmian

Tabela 2: Historia zmian

Autor	Data	Opis zmian	Wersja
Kowalczyk & Piechota	2018-01-07	Stworzenie pierwszej wersji projektu	0.0

PR17wsw503			**Karta oceny projektu**
Data	Ocena	Podpis	Uwagi